Structural Specification-based Testing with ADL

Juei Chang and Debra J. Richardson
{jueic, djr}@ics.uci.edu
Information and Computer Science, University of California, Irvine, CA 92717-3425

Sriram Sankar
sankar@eng.sun.com
Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, CA 94043-1100

Abstract

This paper describes a specification-based black-box technique for testing program units. The main contribution is the method that we have developed to derive *test conditions*, which are descriptions of test cases, from the formal specification of each program unit. The derived test conditions are used to guide test selection and to measure comprehensiveness of existing test suites. Our technique complements traditional code-based techniques such as statement coverage and branch coverage. It allows the tester to quickly develop a black-box test suite.

In particular, this paper presents techniques for deriving test conditions from specifications written in the Assertion Definition Language (ADL) [SH94], a predicate logic-based language that is used to describe the relationships between inputs and outputs of a program unit. Our technique is fully automatable, and we are currently implementing a tool based on the techniques presented in this paper.

1 Introduction

Structural testing usually refers to techniques where test cases are intended to cover some structure of the implementation. The technique that we present in this paper is a structural specification-based testing technique, where test cases are intended to cover some structure of the specification. These specification-based test cases are interesting because they relate directly to what the program is supposed to do and can detect certain errors (in particular, missing path errors) that sometimes are not detected by implementation-based testing.

Test selection is an activity that attempts to partition the input and the output domains of the program into a finite number of subdomains that are approximations of equivalence classes. In our approach, we use *test conditions* to characterize each subdomain. A test condition evaluates to true only for test data that are members of the subdomain associated with that test condition.

In this paper, we discuss how structural specification-based test conditions can be derived from ADL (Assertion Definition Language) specifications. One uses ADL to describe the behavior of a program unit (a procedure or a function). An ADL specification consists of a set of assertions that must hold immediately after the termination of any call to the specified program unit. ADL assertions are based on first order predicate logic. Each assertion is a boolean expression that constrains values of input and output parameters of the specified unit.

Two kinds of test conditions can be derived from ADL specifications: *call-state test conditions* and *return-state test conditions*. Call-state test conditions are test conditions that are derived from the input conditions of the function as described in the specification and constrain values of input parameters only. They characterize subdomains that are partitions of a function's input domain. Return-state test conditions are derived from both the input and output conditions of the function and constrain values of both input and output parameters. They characterize subdomains that are partitions of both input and output domains. Call-state test conditions can be used to measure comprehensiveness of test data even if an implementation is not available. This adheres to the sound principle of developing the black-box test suite during design and not after the implementation has been developed. Return-state test conditions provide more thorough test coverage, but cannot be evaluated without an implementation.

The ADL Translator (ADLT) provides automated support for testing C programs. Given the ADL specification of a C function and specifications of user selected test data (Test Data Description), ADLT generates a *test driver* that executes the *function*

This work is sponsored in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency, under Contract Number F30602-94-C-0218, by Hughes Aircraft Company and the University of California under MICRO Grant Number 94-105, and by a gift from Sun Microsystems Laboratories. The content does not necessarily reflect the position or the policy of the U.S. Government, Hughes Aircraft Company, Sun Microsystems Laboratories, or the University of California, and no official endorsement should be inferred.

ISSTA '96, San Diego CA USA
© 1996 ACM 0-89791-787-1/95/01..$3.50

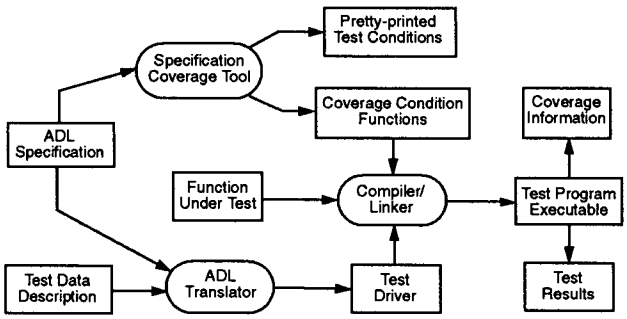

Figure 1: ADL Translator and Specification Coverage Tool

under test with those test data and automatically checks test results.

We are currently developing a tool, called the Specification Coverage Tool (SCT), that derives test conditions from ADL specifications. Figure 1 shows how SCT is integrated with ADLT. SCT generates *coverage condition functions*. Coverage condition functions are C functions that determine whether the derived test conditions are satisfied by some test data. Coverage condition functions are compiled and linked with the ADLT-generated test driver and the function under test. The compiled and linked executable is called the *test program executable*. During testing, the test program executable collects coverage information and updates the *coverage information file*. The coverage information file records the number of times each coverage condition is satisfied. SCT also generates a pretty-printed listing of derived test conditions. This listing can be used by the tester to develop actual test inputs.

2 Summary of ADL

We shall use an Elevator example to illustrate some of the constructs of ADL. Figure 2 shows an ADL specification of part of an elevator system. The part that we have chosen to specify is a function that computes which floor the elevator should move to next, given its current direction, current location, and pending calls and requests. For our purpose, a *call* for the elevator is made on a floor to move either up or down. A *request* is made inside the elevator to visit a particular floor.

An ADL specification consists of a set of *modules*. Each module encapsulates a set of *constituents* that describe the entities in the specified program. A module may import other modules. The constituents of an imported module are visible to the importing module. In the Elevator example, the `elevator` module imports the `floors` module. The `floors` module specifies a *set* abstract data type (ADT). The operations of the `floors` module are described in the Appendix.

There are three types of constituents: *type constituents*, *object constituents*, and *function constituents*. A type constituent defines a type. An object constituent introduces an object of some type. A function constituent introduces a function with its parameters and the return type. The `elevator` module has these following constituents: (1) `INVALID_FLOOR` and `INVALID_CALL` are constants of type `int`, (2) `error` is an object of type `int`, (3) `direction` is an enumeration type that has two enumeration constants: `UP` and `DOWN`, (4) `moveElevator` is a function constituent.

Function constituents may contain *semantic descriptions*. The semantic description of a function constituent describes the function's behavior. There are two types of semantic descriptions: *bindings* and *assertions*. Bindings bind names to expressions. The names then represent abbreviations for their associated expressions. The names `normal` and `exception` have special meanings in ADL, that is, the expressions that they are bound to characterize the normal and exceptional behaviors of the function. The first binding of `moveElevator` binds the expression `(return == -1)` to the name `exception`, and the second binding binds `!exception` to `normal`.

An assertion is a boolean expression that must evaluate to true at the termination of function execution. An assertion is a post-condition of the function. An important predefined ADL operator is the call-state operator ("@") that takes an expression as an argument and evaluates it at the *call state*. The call state refers to the time an implementation of the specified function is called. Similarly, the *return state* refers to the time of return from a call to an implementation of the function. Another important ADL expression is the exception expression $(p <:> q)$. An exception expression prescribes an exceptional outcome of the specified function and is defined as follows (here "-->" is the ADL logical implication operator): p `<:>` q is equivalent to

```
(p --> exception) &&
((exception && q) --> p)
```

We shall use the term *exception pre-conditions* to refer to the left operands of exception expressions.

The first two assertions of `moveElevator` describe its exceptional behavior. The first exception expression (Assertion 1) represents the situation where the current location of the elevator is above the highest floor or lower than the bottom floor. The second exception expression (Assertion 2) represents the situation where an up call is made on the highest floor or a down call is made on the lowest floor. The specification therefore requires that in either of these cases, `moveElevator` behaves exceptionally, that is, it returns `-1`. In addition, the specification also prescribes that `error` be set to `INVALID_FLOOR` if the first exception pre-condition is true, and that `error` be set to `INVALID_CALL` if the second exception pre-condition is true. The specification also requires that if `error` is equal to `INVALID_FLOOR` after a function call that returns `-1`, the first exception pre-condition must be true. Similarly, if `error` is equal to `INVALID_CALL` after a function call that returns `-1`, the second exception pre-condition must be true.

The normal behavior of the function is specified by the normally expression. It has the following form:

```
normally { e1, e2, ... }
```

```
module elevator imports floors {

 const int INVALID_FLOOR;
 const int INVALID_CALL;
 int error;
 typedef enum direction { UP, DOWN } direction;

 int moveElevator(
  direction *currentDirection,
  floor *currentFloor,
  floorSet requests,
  floorSet upCalls,
  floorSet downCalls)

 semantics {

  exception := (return == -1),
  normal := !exception,

  prevFloor := @(*currentFloor),
  prevRequests := @(duplicate(requests)),
  prevUpCalls := @(duplicate(upCalls)),
  prevDownCalls := @(duplicate(downCalls)),

  /* Assertion 1 */
  @(*currentFloor > MAX_FLOOR
    || *currentFloor < 1)
  <:> error == INVALID_FLOOR,

  /* Assertion 2 */
  @(isMember(MAX_FLOOR, upCalls)
    || isMember(1, downCalls))
  <:> error == INVALID_CALL,

  normally {

  /* Assertion 3 */
  @(*currentDirection == UP
    && *currentFloor
      <= max(setUnion(requests, upCalls)))
  -->
  (*currentDirection == UP
   && *currentFloor ==
     min(deleteElements(setUnion(prevRequests,
       prevUpCalls), 1, prevFloor - 1))
   && equal(requests, deleteElement(prevRequests,
     *currentFloor))
   && equal(upCalls, deleteElement(prevUpCalls,
     *currentFloor))
   && equal(downCalls, prevDownCalls)),

  /* Assertion 4 */
  @(*currentDirection == DOWN
    && *currentFloor
      >= min(setUnion(requests, downCalls)))
  -->
  (*currentDirection == DOWN
   && *currentFloor ==
     max(deleteElements(setUnion(prevRequests,
       prevDownCalls),prevFloor + 1, MAX_FLOOR))
   && equal(requests, deleteElement(prevRequests,
     *currentFloor))
   && equal(downCalls,deleteElement(prevDownCalls,
     *currentFloor))
   && equal(upCalls, prevUpCalls)),


  /* Assertion 5 */
  @(*currentDirection == UP
    && *currentFloor >
      max(setUnion(requests, upCalls))
    && !empty(setUnion(requests, downCalls)))
  -->
  (*currentDirection == DOWN
   && *currentFloor == max(setUnion(prevRequests,
     prevDownCalls))
   && equal(requests,
     deleteElement(prevRequests, *currentFloor))
   && equal(upCalls, prevUpCalls)
   && equal(downCalls,deleteElement(prevDownCalls,
     *currentFloor))),

  /* Assertion 6 */
  @(*currentDirection == DOWN
    && *currentFloor
      < min(setUnion(requests, downCalls))
    && !empty(setUnion(requests, upCalls)))
  -->
  (*currentDirection == UP
   && *currentFloor == min(setUnion(prevRequests,
     prevUpCalls))
   && equal(requests, deleteElement(prevRequests,
     *currentFloor))
   && equal(upCalls, deleteElement(prevUpCalls,
     *currentFloor))
   && equal(downCalls, prevDownCalls)),

  /* Assertion 7 */
  @(*currentDirection == UP
    && !empty(upCalls)
    && *currentFloor > max(upCalls)
    && empty(setUnion(downCalls, requests)))
  -->
  (*currentDirection == UP
   && *currentFloor == min(prevUpCalls)
   && equal(requests, prevRequests)
   && equal(upCalls, deleteElement(prevUpCalls,
     *currentFloor))
   && equal(downCalls, prevDownCalls)),

  /* Assertion 8 */
  @(*currentDirection == DOWN
    && !empty(downCalls)
    && *currentFloor < min(downCalls)
    && empty(setUnion(upCalls, requests)))
  -->
  (*currentDirection == DOWN
   && *currentFloor == max(prevDownCalls)
   && equal(requests, prevRequests)
   && equal(upCalls, prevUpCalls)
   && equal(downCalls,
     deleteElement(prevDownCalls, *currentFloor)))
  }
 };
};
```

Figure 2: ADL specification of the elevator module.

Each of the e_i's must be true for all function calls that behave normally. For `moveElevator`, there are six assertions in the normally expression. The specification states that if `moveElevator` behaves normally, that is, it returns anything other than `-1`, all six assertions must be true.

Here we provide an informal description of the elevator's normal behavior to help the reader understand the Elevator specification. The elevator will alternate upward and downward cycles. In a upward cycle, the elevator will move up to the nearest higher floor with an outstanding request or up call (Assertion 3). It will repeat this until there is no longer a higher floor for a request or an up call. If there are down calls or requests for a lower floor, it will change direction (Assertion 5) and will then execute an analogous downward cycle (Assertion 4). It may also be that there are no requests and the only outstanding calls are up calls from lower floors. Since a downward cycle would not service these, the elevator will travel to the lowest floor having such an up call (Assertion 7) and then behave according to the rules for an upward cycle. A dual situation exists at the end of a downward cycle and is handled analogously (Assertions 6 and 8).

3 Terminology

A *test condition* is a set of boolean conditions that are constraints on values of parameters — input parameters, output parameters, global variables, and the return value — of the function under test. A *call-state test condition* is a test condition that constrains values of input parameters and input values of global variables. Below is a call-state test condition for the Elevator example:

```
{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor >= 1),
  @(!(isMember(MAX_FLOOR, upCalls))),
  @(!(isMember(1, downCalls))),
  @(*currentDirection == UP),
  @(*currentFloor
      > max(setUnion(requests, upCalls))) }
```

This test condition represents a condition that some test data should satisfy. It can be used by a tester to select test data that satisfy this condition. It can also be used by tools to automatically determine whether this condition is satisfied by some test suite. For example, assuming `MAX_FLOOR` is greater than 4, a tester can select the following test data to satisfy this test condition:

```
@(*currentDirection): UP
@(*currentFloor): 4
@(requests): { 1 }
@(upCalls): { 3 }
@(downCalls): { 2, 5 }
```

As will be discussed in more details later, call-state test conditions are derived from *call-state evaluatable expressions* in ADL specifications. A call-state evaluatable expression is an expression where the values of variables in the expression either cannot change during the execution of the function or are evaluated before the function is called. In C, all parameters are passed by value. Thus, the value of any parameter cannot change[1]. However, the value of an object pointed by a pointer parameter can change. Elements of an array parameter can change. Function calls may also modify values of global variables. Therefore, any subexpression of an assertion that does not contain any pointers, arrays, function calls, or global variables (except constants) is a call-state evaluatable subexpression of that assertion. Also, the operands of the call-state operator ("`@`") is evaluated before the function call. Thus, they are also call-state evaluatable.

A *return-state test condition* is a test condition that constrains values of input parameters, output parameters, global variables, and the return value. Below is a return-state test condition for the Elevator example:

```
{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor >= 1),
  @(!(isMember(MAX_FLOOR, upCalls))),
  @(!(isMember(1, downCalls))),
  return != -1,
  @(*currentDirection == UP),
  @(*currentFloor
      <= max(setUnion(requests, upCalls))),
  *currentDirection == UP,
  *currentFloor == min(deleteElements(
    setUnion(@(duplicate(requests)),
      @(duplicate(upCalls))), 1,
        @(*currentFloor) - 1)),
  equal(requests,deleteElement(
    @(duplicate(requests)), *currentFloor)),
  equal(upCalls, deleteElement(
    @(duplicate(upCalls)), *currentFloor)),
  equal(downCalls, @(duplicate(downCalls))) }
```

For example, a test case with the following test input values and output values would satisfy this test condition:

```
@(*currentDirection): UP
@(*currentFloor): 3
@(requests): { 2 }
@(upCalls): { 1, 4 }
@(downCalls): { 3, 4 }
return: 0
*currentDirection: UP
*currentFloor: 4
requests: { 2 }
upCalls: { 1 }
downCalls: { 3, 4 }
```

4 Approach

In short, our approach derives test conditions by traversing the parse tree of the specification, collecting and combining boolean conditions from various parts of the specification. The methods for generating and combining these boolean conditions are called *rules*. Rules are associated with the types of nodes in the parse tree and are based on various test selection strategies.

To generate call-state test conditions, rules are only applied to call-state evaluatable expressions in the specification. To generate return-state test conditions, rules are applied to all expressions.

---

[1] Actually this not entirely true. If a parameter is aliased (referenced by another parameter or a global variable), the value of the parameter could change. We do not consider this situation in this paper.

4.1 Test Selection Strategies

The ADL constructs that are most relevant to testing are logical expressions (*e.g.* "&&", "||", "-->") and relational expressions (*e.g.* ">", ">=", "<", "<="). For logical expressions, rules based on the multicondition strategy [Mye79] and the meaningful impact strategy [Fos84, WGS94] can be used. The multicondition strategy selects every possible value for each operand of a logical expression. The meaningful impact strategy requires each logical operand to individually affect the value of the expression. Rules for relational expressions are based on boundary value strategy [Mye79] and domain testing strategies [WC80, CHR82]. These strategies select test points on or near a boundary.

Rules can be developed for other constructs also. For a particular application, strategies that are suitable for that application can be developed. For example, rules that are based on computation testing strategies [Fos80, CR83] can be used on mathematical expressions to test scientific or numerical programs.

For the Elevator example, we shall use the multicondition strategy for logical expressions. For clarity, we provide two examples of the multicondition strategy here. For the expression `(a || b)`, the multicondition strategy produces the following conditions: `{!a, !b}`, `{!a, b}`, and `{a}`. For the expression `(a --> b)`, the following conditions are produced: `{!a}`, `{a, !b}` and `{a, b}`. Our approach uses a top-down tree traversal algorithm where each parse tree node may have an inherited attribute. This attribute is a constraint on the value of the expression associated with the node. When such a constraint is present, only the cases, as required by the multicondition strategy, that are consistent with the constraint are generated. For example, we might require the value of `(a || b)` be true, the multicondition strategy would yield only the cases that make the expression true: `{!a, b}` and `{a}`. If the value of `(a || b)` is constrained to false, only one case is generated: `{!a, !b}`.

For relational expressions, we shall use a simple boundary testing strategy. This strategy selects test points on both sides of a boundary. For example, for `(a > b)`, this strategy requires: `{a == b + 1}` and `{a == b}`. If there is a boolean constraint on the value of the expression, only cases that are consistent with the constraint will be generated. For example, if `(a > b)` is constrained to true, assuming *a* and *b* are integers, one condition, `{a == b + 1}` is generated. If the value of `(a > b)` is constrained to false, only `{a == b}` is generated.

4.2 Generating Test Conditions

ADL allows the specification of both the normal behavior and the exceptional behavior of the function under test. Both of these behaviors should be tested.

4.2.1 Testing Normal Behavior

To test the normal behavior, `exception` must have the value `false`. Also, all the exception pre-conditions must be suppressed, that is, for all exception expressions p_i <:> q_i, we require that all p_i's must have the value `false`. In addition, each expression e_i in the normally expression, `normally { e1, e2, ... }`, should have the value `true`.

In the Elevator example, `exception` is bound to the expression `(return == -1)`. The multicondition strategy requires all cases that would make this expression false. The boundary testing strategy does not apply since there is no relational operator in this expression. Our algorithm traverses the parse tree of this expression in a top down manner and passes the "false" constraint down the tree as a parameter. Upon reaching the equality node ("=="), the algorithm generates a boolean condition, M_1, to make the equality "false":

```
{ return != -1 }.                                   (M1)
```

As mentioned earlier, we require that all the exception pre-conditions, p_i's, must be false.The p_i's in the Elevator example are:

```
@(*currentFloor > MAX_FLOOR
  || *currentFloor < 1)
```

and

```
@(isMember(MAX_FLOOR, upCalls)
  || isMember(1, downCalls)).
```

For the first expression, the algorithm traverses the parse tree of that expression top-down with the "false" constraint. Upon reaching the logical OR node ("||"), the multicondition strategy requires that both the left-hand side and the right-hand side must be false. Thus, the algorithm traverses the left subtree with the "false" constraint. It then traverses the right subtree also with the "false" constraint. For the left subtree, requiring the value of `(*currentFloor > MAX_FLOOR)` be false, the algorithm generates the following condition:

```
{ @(*currentFloor <= MAX_FLOOR) }.                  (M2)
```

Requiring the expression be false, the boundary testing strategy generates the following boundary condition B_1:

```
{ @(*currentFloor == MAX_FLOOR) }.                  (B1)
```

Requiring the value of the right subtree `(*currentFloor < 1)` be false, the algorithm generates:

```
{ @(*currentFloor >= 1) }                           (M3)
```

and the boundary condition:

```
{ @(*currentFloor == 1) }.                          (B2)
```

We use the same algorithm to generate conditions from the second exception expression:

```
{ @(!isMember(MAX_FLOOR, upCalls)) }                (M4)
```

and

```
{ @(!isMember(1, downCalls)) }.                     (M5)
```

Note that no boundary conditions are generated from the second exception expression.

Each expression e_i in the normally expression, `normally { e1, e2, ... }`, should have the value `true`. For each e_i, we use the same top-down tree traversal to generate conditions. For Assertion 3 of the Elevator example, the multicondition strategy requires the following conditions:

```
{ @(*currentDirection != UP) },                     (M6)
```

```
{ @(*currentDirection == UP),
  @(*currentFloor >
    max(setUnion(requests, upCalls))) },        (M7)
```

and

```
{ @(*currentDirection == UP),
  @(*currentFloor <=
    max(setUnion(requests, upCalls))),
  *currentDirection == UP,
  *currentFloor ==
    min(deleteElements(setUnion(prevRequests,
      prevUpCalls), 1, prevFloor - 1)),
  equal(requests, deleteElement(prevRequests,
    *currentFloor)),
  equal(upCalls, deleteElement(prevUpCalls,
    *currentFloor)),
  equal(downCalls, prevDownCalls))
}.                                              (M8)
```

M_6 and M_7 represent all possible cases that make the left-hand side of the implication expression of Assertion 3 false. M_8 represents the case where both the left-hand side and the right-hand side of the implication expression are true.

The boundary testing strategy requires the following conditions for Assertion 3:

```
{ @(*currentDirection == UP),
  @(*currentFloor ==
    max(setUnion(requests, upCalls)) + 1)},     (B3)
```

```
{ @(*currentDirection == UP),
  @(*currentFloor ==
    max(setUnion(requests, upCalls))),
  *currentDirection == UP,
  *currentFloor ==
    min(deleteElements(setUnion(prevRequests,
      prevUpCalls), 1, prevFloor - 1)),
  equal(requests, deleteElement(prevRequests,
    *currentFloor)),
  equal(upCalls, deleteElement(prevUpCalls,
    *currentFloor)),
  equal(downCalls, prevDownCalls))}.            (B4)
```

Finally, we combine the conditions generated from the expression bound to `exception` and the exception pre-conditions with the boolean conditions generated from the assertion in the normally expression. For Assertion 3 of the Elevator example, the multicondition strategy requires three return-state test conditions (T_1, T_2, and T_3):

$$T_1 = M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5 \cup M_6,$$
$$T_2 = M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5 \cup M_7,$$
$$T_3 = M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5 \cup M_8.$$

For clarity, we show the conditions associated with T_1:

```
{ return != -1,
  @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor >= 1),
  @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentDirection != UP) }.
```

We can generate two kinds of boundary test conditions. Both represent boundary cases of the normal behavior. Test conditions of the first kind are obtained by combining the boundary conditions generated from the normally expression with the non-boundary conditions generated from the expression bound to `exception` and the exception pre-conditions. For Assertion 3, the boundary testing strategy requires these return-state test conditions:

$$T_4 = M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5 \cup B_3,$$
$$T_5 = M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5 \cup B_4.$$

We can also combine boundary conditions generated from the exception pre-conditions and the expression bound to `exception` with non-boundary conditions generated from the normally expression. For Assertion 3, the following return-state test conditions are required:

$$T_6 = M_1 \cup B_1 \cup M_3 \cup M_4 \cup M_5 \cup M_6,$$
$$T_7 = M_1 \cup B_1 \cup M_3 \cup M_4 \cup M_5 \cup M_7,$$
$$T_8 = M_1 \cup B_1 \cup M_3 \cup M_4 \cup M_5 \cup M_8,$$
$$T_9 = M_1 \cup M_2 \cup B_2 \cup M_4 \cup M_5 \cup M_6,$$
$$T_{10} = M_1 \cup M_2 \cup B_2 \cup M_4 \cup M_5 \cup M_7,$$
$$T_{11} = M_1 \cup M_2 \cup B_2 \cup M_4 \cup M_5 \cup M_8,$$

Note that in the above test conditions, each boundary is tested separately. We could also generate additional test conditions, each with multiple boundaries, but this would substantially increase the number of test conditions. Also we could have generated fewer boundary test conditions if we have chosen not to generate boundary conditions from exception pre-conditions. Whether to generate more or fewer test conditions depends on what level of rigor is required and also on how much resource is allocated to testing.

The call-state test conditions are obtained exactly the same way except that only call-state evaluatable expressions are considered. In the Elevator example, the expression bound to `exception` is not call-state evaluatable. Thus, no condition is generated from it. The conditions that are derived from exception pre-conditions are the same as those generated for return-state analysis (M_2, M_3, M_4, M_5, B_1, and B_2) since both exception pre-conditions are call-state evaluatable. Since only the left operand of the implication expression of Assertion 3 is call-state evaluatable, only the left operand is considered. The following conditions are required by the multicondition strategy:

```
{ @(*currentDirection != UP) },
```

```
{ @(*currentDirection == UP),
  @(*currentFloor >
    max(setUnion(requests, upCalls))) },
```

and

```
{ @(*currentDirection == UP),
  @(*currentFloor <=
    max(setUnion(requests, upCalls))) }.
```

The following conditions are required by the boundary testing strategy:

```
{ @(*currentDirection == UP),
  @(*currentFloor ==
    max(setUnion(requests, upCalls)) + 1) },
```

and

```
{ @(*currentDirection == UP),
  @(*currentFloor ==
    max(setUnion(requests, upCalls))) }.
```

The conditions are combined in the same way as in return-state analysis. For example, the call-state test conditions generated using only the multicondition strategy for Assertion 3 of the Elevator example are shown below:

```
{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor >= 1),
  @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentDirection != UP) },

{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor >= 1),
  @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentDirection == UP),
  @(*currentFloor >
    max(setUnion(requests, upCalls))) },
```

and

```
{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor >= 1),
  @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentDirection == UP),
  @(*currentFloor <=
    max(setUnion(requests, upCalls))) }.
```

The method that we have described also applies to Assertions 4-8 of the Elevator example.

4.2.2 Testing Exceptional Behavior

To test the specified exceptional behavior of the function, `exception` must have the value `true`. In the Elevator example, requiring `exception` be true produces

```
{ (return == -1) }                                   (M9)
```

To test the exceptional behavior prescribed by an exception expression p_i <:> q_i in the specification, we require that all other exception pre-conditions p_j's in p_j <:> q_j take the value `false` and require both p_i and q_i be `true`.

To test the behavior prescribed by Assertion 1 of the Elevator example, the exception pre-condition of Assertion 2 must be false. Applying the multicondition strategy to the pre-condition of Assertion 2, requiring its value be false yields:

```
{ @(!isMember(MAX_FLOOR, upCalls)) }                 (M10)
```

and

```
{ @(!isMember(1, downCalls)) }.                      (M11)
```

Requiring both operands of Assertion 1 be true, the multicondition strategies requires:

```
{ @(*currentFloor > MAX_FLOOR),
  error == INVALID_FLOOR },                          (M12)

{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor < 1),
  error == INVALID_FLOOR },                          (M13)
```

Applying the boundary testing strategy yields the following boundary conditions:

```
{ @(*currentFloor == MAX_FLOOR + 1),
  error == INVALID_FLOOR },                          (B5)

{ @(*currentFloor == MAX_FLOOR),
  @(*currentFloor < 1),
  error == INVALID_FLOOR },                          (B6)

{ @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor == 1 - 1),
  error == INVALID_FLOOR }.                          (B7)
```

Our algorithm then combines conditions generated from the exception expression being considered, from other exception expressions (p_j <:> q_j), and from the expression bound to `exception`. Two return-state test conditions are generated using the multicondition strategy on Assertion 1 of the Elevator example:

$$T_{12} = M_9 \cup M_{10} \cup M_{11} \cup M_{12},$$

$$T_{13} = M_9 \cup M_{10} \cup M_{11} \cup M_{13}.$$

Three return-state test conditions are generated using the boundary testing strategy on Assertion 1:

$$T_{14} = M_9 \cup M_{10} \cup M_{11} \cup B_5,$$

$$T_{15} = M_9 \cup M_{10} \cup M_{11} \cup B_6,$$

$$T_{16} = M_9 \cup M_{10} \cup M_{11} \cup B_7.$$

Call-state test conditions can be obtained using the same method except that only call-state evaluatable expressions are considered. Call-state test conditions for Assertion 1 of the Elevator example are:

```
{ @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentFloor > MAX_FLOOR) },

{ @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor < 1) },

{ @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentFloor == MAX_FLOOR + 1) },

{ @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentFloor == MAX_FLOOR),
  @(*currentFloor < 1) },
```

and

```
{ @(!isMember(MAX_FLOOR, upCalls)),
  @(!isMember(1, downCalls)),
  @(*currentFloor <= MAX_FLOOR),
  @(*currentFloor == 1 - 1) }.
```

The method we have described also applies to Assertion 2.

4.2.3 Elevator Results

Using the multicondition strategy, our algorithm would generate a total of 28 call-state test conditions and 28 return-state test conditions[2] from the Elevator specification. These test conditions are all feasible.

Applying the boundary testing strategy to the normally expression, our algorithm would generate 16 call-state test conditions and 16 return-state test conditions. These test conditions are all feasible also.

Applying the boundary testing strategy to exception pre-conditions and requiring `exception` take the value `true`, our algorithm would generate 3 call-state and 3 return-state test conditions. One call-state test condition and one return-state test condition (T_{15} in Section 4.2.2) are infeasible. Both conditions require `*currentFloor` be equal to `MAX_FLOOR` and be less than `1`.

Applying the boundary testing strategy to exception pre-conditions and requiring `exception` take the value `false`, our algorithm generates 48 call-state and 48 return-state test conditions. Six call-state and six return-state test conditions are infeasible.

5 Related Work

Many papers have focused on specification-based testing. Goodenough and Gerhart [GG76] and Gourlay [Gou83] demonstrate the importance of specification-based testing. Many papers have since focused on deriving tests from the specification. Weyuker and Ostrand [WO80] develop theories of program testing using revealing subdomains. They emphasize the importance of deriving test cases from both the specification and the implementation. However, they do not provide a systematic way for creating the specification partition. Richardson and Clarke [RC85], Cartwright [Car81], and Richardson, O'Malley, and Tittle [ROT89] all propose using symbolic execution techniques to create the specification partition. However, symbolic execution techniques cannot easily be applied to several languages, ADL being one of them. Our technique is an alternative approach that does not require symbolic execution. Several other papers are also related. Ostrand and Balcer [OB88] provide a methodology, the Category-partition Method, for developing functional tests from informal system-level specifications. Stocks and Carrington [SC93] introduce the Test Template Framework in which specification-based testing can be conducted. Whereas their approach focuses on providing a framework for deriving specification-based tests, the focus of our approach is on actually deriving the test conditions. Chang, Sankar, and Richardson [CSR95] present some early ideas on deriving tests from ADL. This paper is the continuation of that work. Doong and Sankar [DS95] describe an implementation of a coverage analyzer for measuring multicondition coverage based on ADL. We adopted a few ideas from this work, in particular, on how exception expressions are handled.

[2] It so happens that for the Elevator example, there is a one-to-one correspondence between a call-state test condition and a return-state condition. Thus, the number of call-state test conditions is the same as the number return-state conditions. In general, this is not the case.

6 Conclusion and Future Work

Having the ability to automatically derive test conditions from formal specifications offers several benefits. It provides a systematic method for developing a black-box test suite. It also provides a way to measure coverage of test data with respect to the specification. We believe that having this capability would further encourage the use of formal specifications.

We are currently working on a tool based on the techniques discussed in this paper. As part of future work, we would like to apply our work to industrial applications and experiment with different test selection strategies. In addition, we would like to conduct an experiment to measure code coverage of test data developed from test conditions that are generated by our method.

Bibliography

[CHR82] L. A. Clarke, J. Hassell, and D. J. Richardson, "A close look at domain testing." *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 380-390, July 1982.

[CR83] L. A. Clarke and D. J. Richardson, "A rigorous approach to error-sensitive testing." In *Proceedings of the Sixteenth Hawaii International Conference on System Sciences*, pp. 197-206, January 1983.

[CSR95] J. Chang, S. Sankar, and D. J. Richardson, "Automated test selection from ADL specifications." In *Proceedings of the California Software Symposium (CSS)*, Irvine, California, March 1995.

[Car81] R. Cartwright, "Formal program testing." In *Proc. 8th Annual ACM Principles of Programming Languages Symposium*, 1981.

[DS95] R. Doong and S. Sankar, "Specification-based coverage criteria for ADL." Submitted for publication, 1995.

[Fos80] K. A. Foster, "Error sensitive test case analysis (ESTCA)." *IEEE Trans. Software Eng.*, vol. SE-6, pp. 258-264, May 1980.

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

[GG76] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection." *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 156-173, June 1975.

[Gou83] J. S. Gourlay, "A mathematical framework for the investigation of testing." *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 686-709, November 1983.

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

[OB88] T. J. Ostrand and M. J. Balcer. "The category-partition method for specifying and generating functional tests." *Communications of the ACM*, 31(6), pp. 676-686, June 1988.

[RC85] D. J. Richardson and L. A. Clarke, "Partition analysis: a method combining testing and verification." *IEEE Trans. Software Eng.*, vol. SE-11, no. 12, pp. 1477-1490, December 1985.

[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)*, pages 11-18, Cambridge, Massachusetts, June 1993.

[ROT89] D. J. Richardson, O. O'Malley, and C. Tittle, "Approaches to specification-based testing." In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 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.

[WGS94] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification." *IEEE Trans. Software Eng.*, vol. SE-20, no. 5, pp. 353-363, May 1994.

[WO80] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains." *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 236-246, May 1980.

[WC80] L. J. White and E. I. Cohen, "A domain strategy for computer program testing." *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 247-257, May 1980.

Appendix The floors Module

`isMember(e, s)` returns `true` if `e` is a member of the set `s`. `duplicate(s)` returns a copy of the set `s`. `setUnion(s, t)` returns the union of sets `s` and `t`. `max(s)` returns the largest element of the set `s`. `min(s)` returns the smallest element of `s`. `deleteElement(s, e)` returns a copy of `s` with `e` removed from the copy. `deleteElements(s, a, b)` returns a copy of `s` with elements ranging from `a` to `b` removed. `equal(s, t)` returns `true` if two sets have the same set of elements. `empty(s)` returns `true` if `s` is empty.