

Software Testing and Quality Assurance

Theory and Practice

Chapter 4

Control Flow Testing

- Basic Idea
- Outline of Control Flow Testing
- Control Flow Graph
- Paths in a Control Flow Graph
- Path Selection Criteria
- Generating Test Input
- Containing Infeasible Paths
- Summary

- Two kinds of basic program statements:
 - Assignment statements (Ex. $x = 2*y$;)
 - Conditional statements (Ex. `if()`, `for()`, `while()`, ...)
- Control flow
 - Successive execution of program statements is viewed as flow of control.
 - Conditional statements alter the default flow.
- Program path
 - A program path is a sequence of statements from entry to exit.
 - There can be a large number of paths in a program.
 - There is an (input, expected output) pair for each path.
 - Executing a path requires invoking the program unit with the right test input.
 - Paths are chosen by using the concepts of path selection criteria.
- Tools: Automatically generate test inputs from program paths.

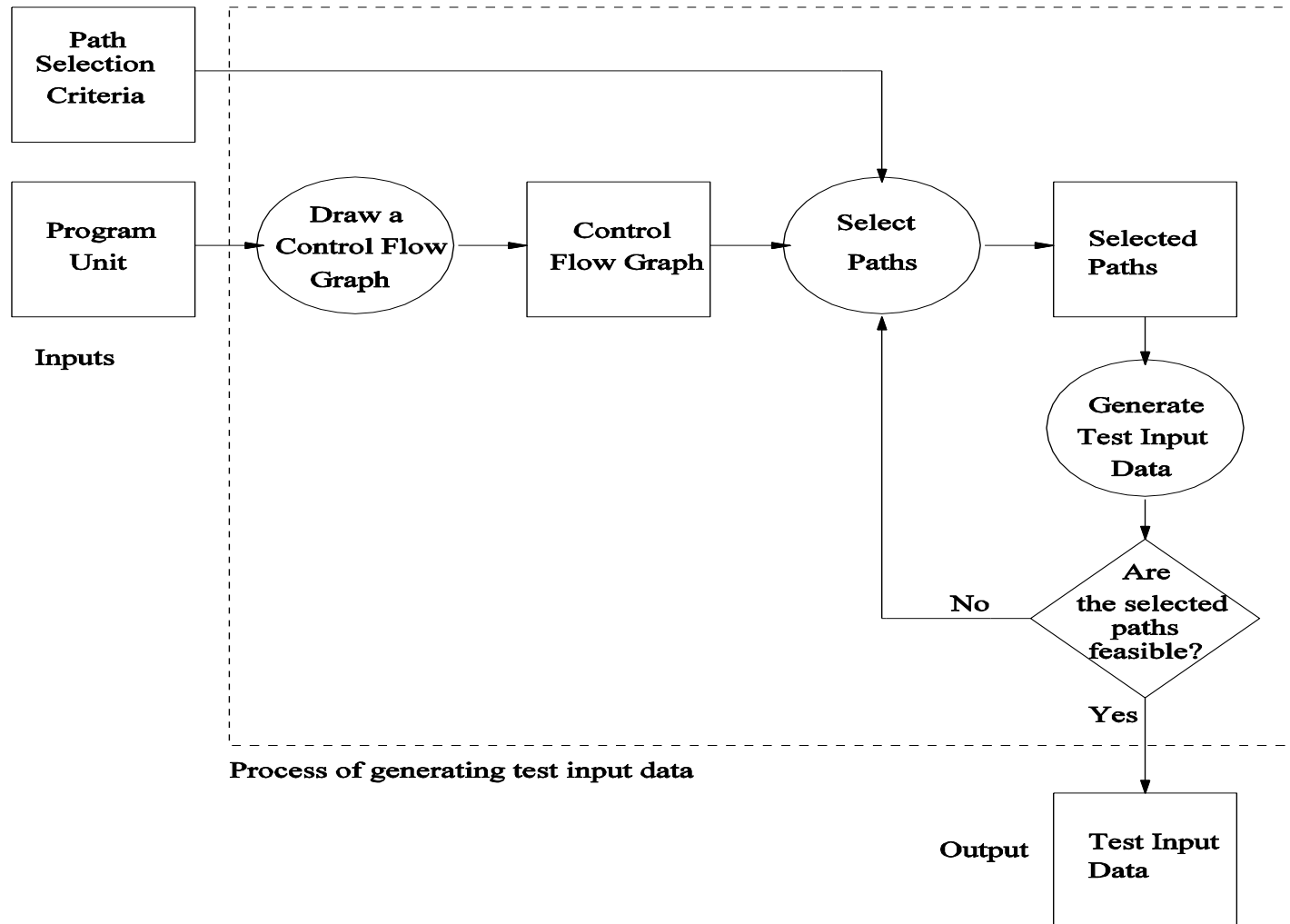


Figure 4.1: The process of generating test input data for control flow testing.

- Inputs to the test generation process
 - Source code
 - Path selection criteria: statement, branch, ...
- Generation of control flow graph (CFG)
 - A CFG is a graphical representation of a program unit.
 - Compilers are modified to produce CFGs. (You can draw one by hand.)
- Selection of paths
 - Enough entry/exit paths are selected to satisfy path selection criteria.
- Generation of test input data
 - Two kinds of paths
 - Executable path: There exists input so that the path is executed.
 - Infeasible path: There is no input to execute the path.
 - Solve the path conditions to produce test input for each path.

- Symbols in a CFG

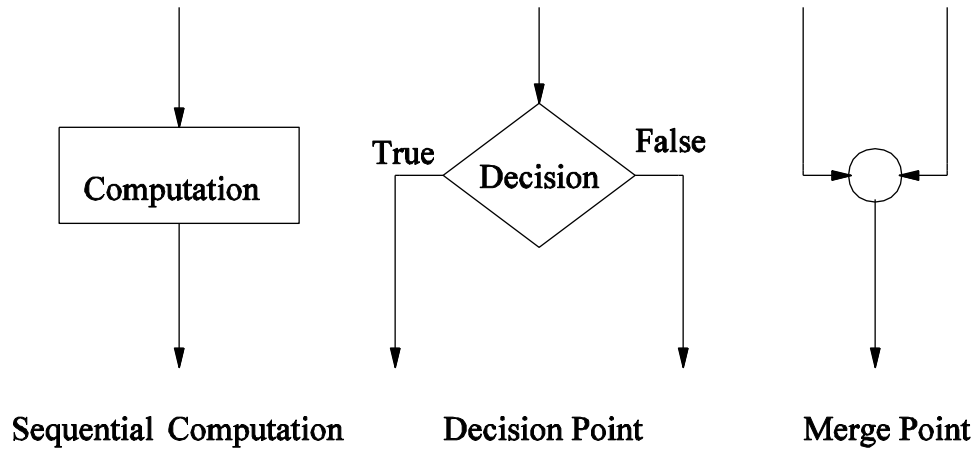


Figure 4.2: Symbols in a control flow graph

- **Example code: openfiles()**

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */
int openfiles(){
    /*
        This function tries to open files "file1", "file2", and "file3"
        for read access, and returns the number of files successfully
        opened. The file pointers of the opened files are put in the
        global variables.
    */
    int i = 0;
    if(
        ((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0)) ||
        ((( fptr2 = fopen("file2", "r")) != NULL) && (i++) && (0)) ||
        ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
    );
    return(i);
}
```

Figure 4.3: A function to open three files.

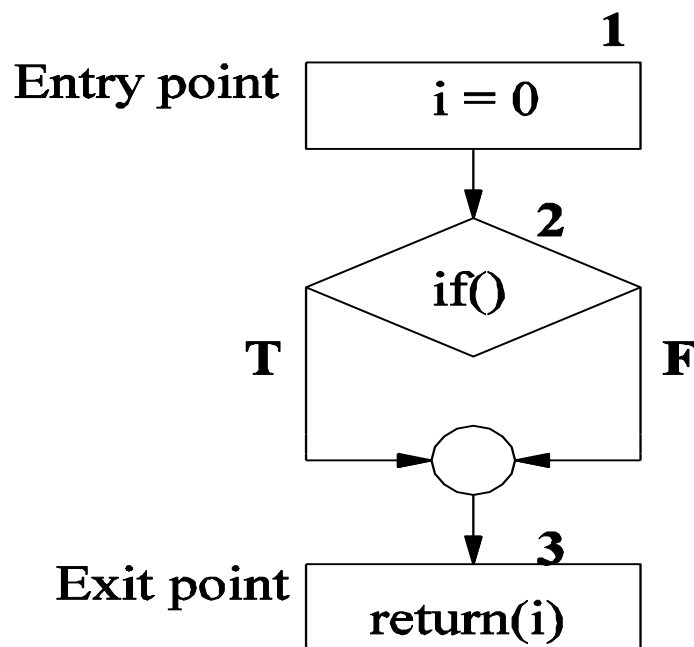


Figure 4.4: A high-level CFG representation of openfiles().

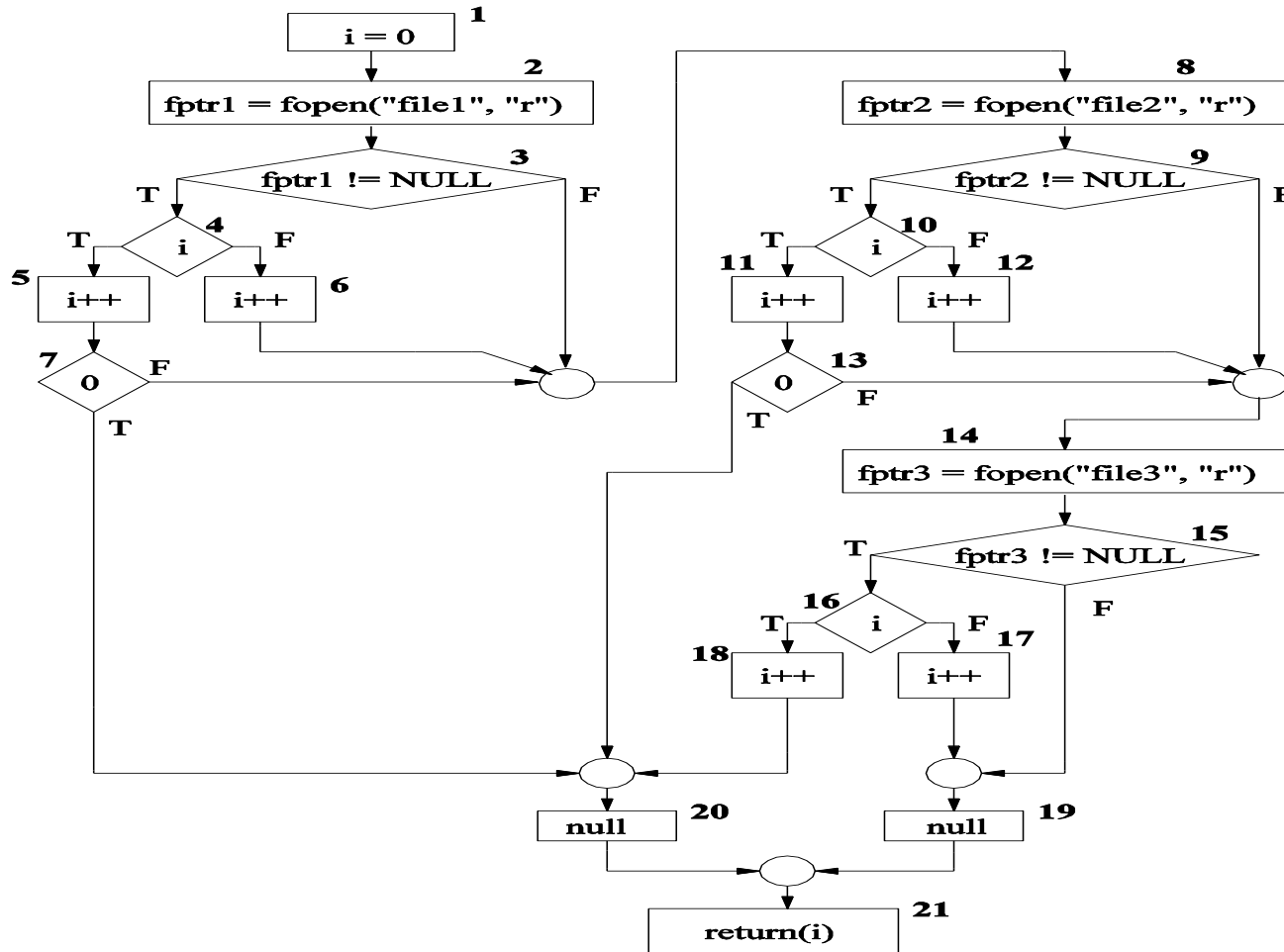


Figure 4.5: A detailed CFG representation of openfiles().

- **Example code: ReturnAverage()**

```
public static double ReturnAverage(int value[], int AS, int MIN, int MAX){
```

```
/* Function: ReturnAverage Computes the average of all those numbers in the input array in
the positive range [MIN, MAX]. The maximum size of the array is AS. But, the array size
could be smaller than AS in which case the end of input is represented by -999. */
```

```
    int i, ti, tv, sum;
    double av;
    i = 0; ti = 0; tv = 0; sum = 0;
    while (ti < AS && value[i] != -999) {
        ti++;
        if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
        }
        i++;
    }
    if (tv > 0)
        av = (double)sum/tv;
    else
        av = (double) -999;
    return (av);
}
```

Figure 4.6: A function to compute the average of selected integers in an array.

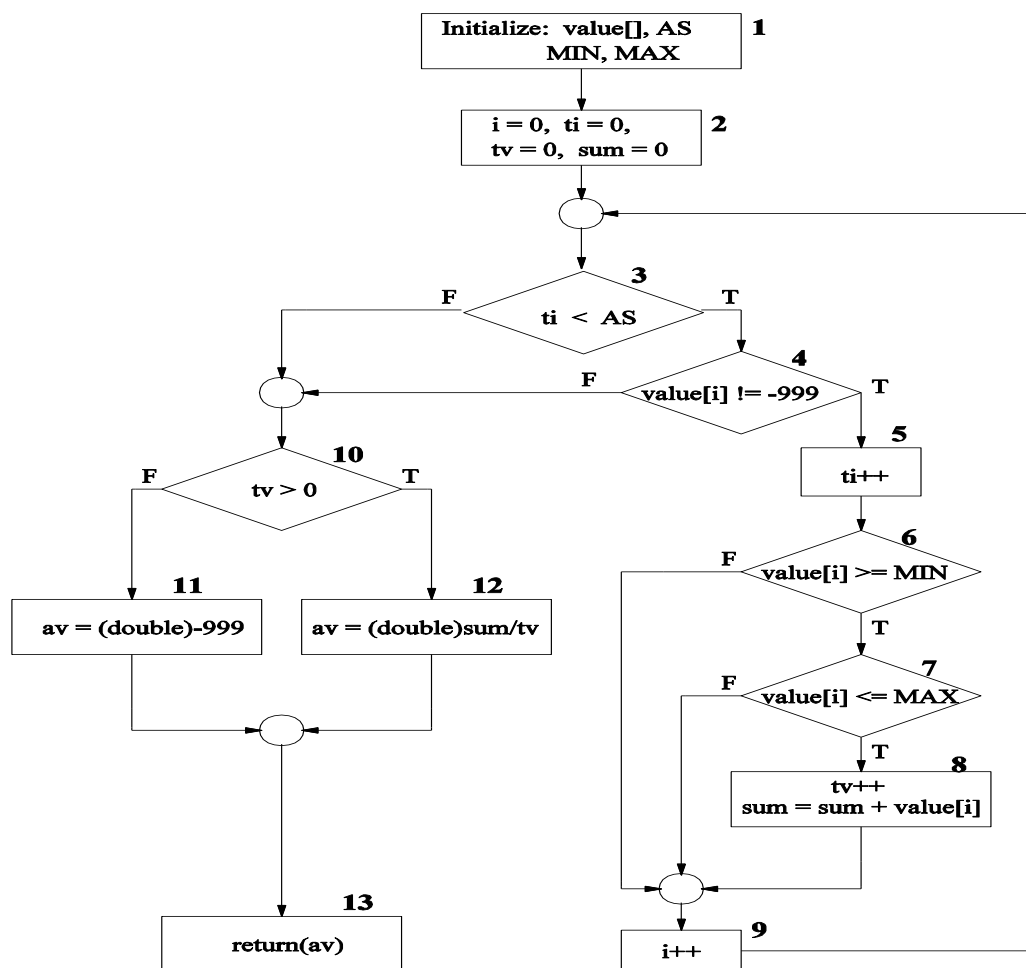


Figure 4.7: A CFG representation of ReturnAverage().

- A few paths in Figure 4.7. (Table 4.1)
 - Path 1: 1-2-3(F)-10(T)-12-13
 - Path 2: 1-2-3(F)-10(F)-11-13
 - Path 3: 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
 - Path 4: 1-2-3(T)-4(T)-5-6-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

- Program paths are selectively executed.
- Question: What paths do I select for testing?
- The concept of *path selection criteria* is used to answer the question.
- Advantages of selecting paths based on defined criteria:
 - Ensure that all program constructs are executed at least once.
 - Repeated selection of the same path is avoided.
 - One can easily identify what features have been tested and what not.
- Path selection criteria
 - Select all paths.
 - Select paths to achieve complete statement coverage.
 - Select paths to achieve complete branch coverage.
 - Select paths to achieve predicate coverage.

- All-path coverage criterion: Select all the paths in the program unit under consideration.
 - The `openfiles()` unit has 25+ paths.

Existence of “file1”	Existence of “file2”	Existence of “file3”
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

Table 4.2: The input domain of `openfiles()`

- Selecting all the inputs will exercise all the program paths.

Input	Path
<No, No, No>	1-2-3(F)-8-9(F)-14-15(F)-19-21
<Yes, No, No>	1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21
<Yes, Yes, Yes>	1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14- 15(T) - 16(T)-18-20-21

Table 4.3 Inputs and paths in openfiles()

- Statement coverage criterion
 - Statement coverage means executing individual program statements and observing the output.
 - 100% statement coverage means all the statements have been executed at least once.
 - Cover all assignment statements.
 - Cover all conditional statements.
 - Less than 100% statement coverage is unacceptable.

SCPath1	1-2-3(F)-10(F)-11-13
SCPath2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Table 4.4: Paths for statement coverage of the CFG of Figure 4.7.

- Branch coverage criterion
 - A branch is an outgoing edge from a node in a CFG.
 - A condition node has two outgoing branches – corresponding to the True and False values of the condition.
 - Covering a branch means executing a path that contains the branch.
 - 100% branch coverage means selecting a set of paths such that each branch is included on some path.

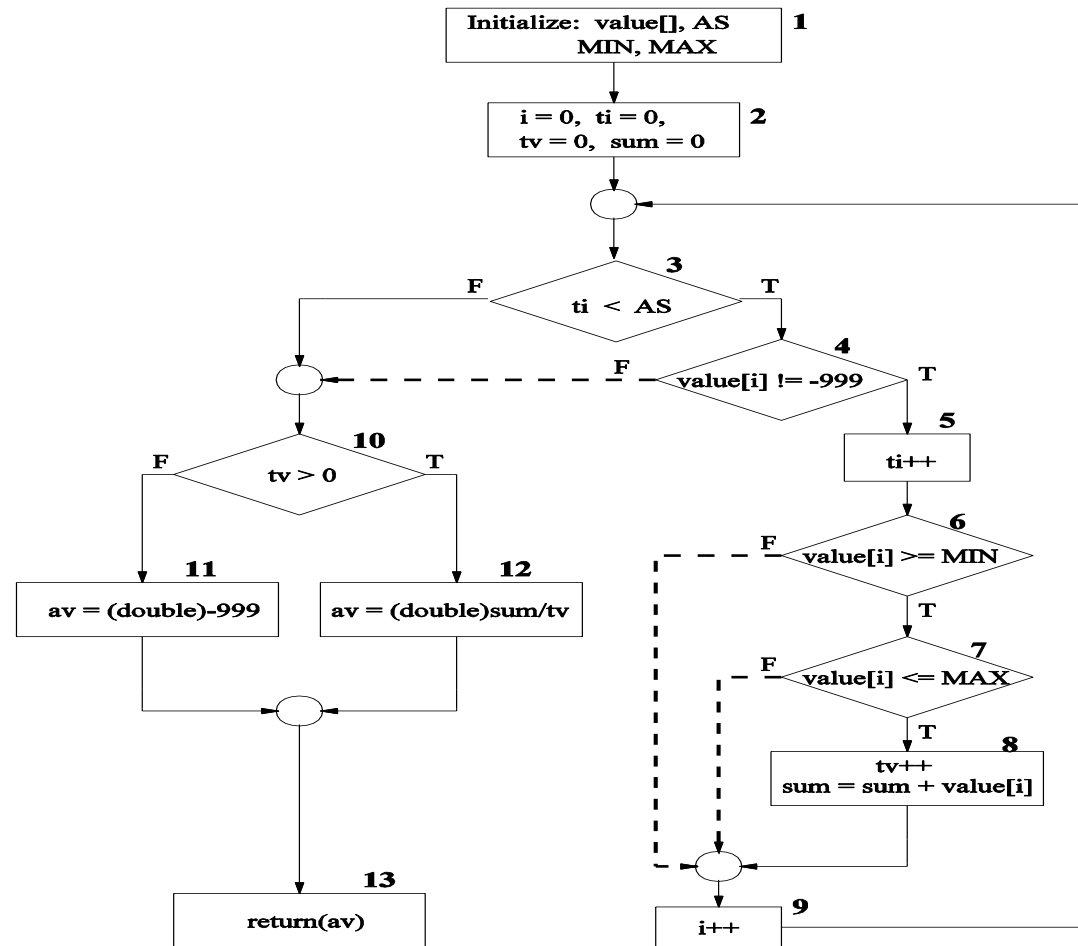


Figure 4.8: The dotted arrows represent the branches not covered by the statement covering in Table 4.4.

- Branch coverage criterion
 - A branch is an outgoing branch (edge) from a node in a CFG.
 - A condition node has two outgoing branches – corresponding to the True and False values of the condition.
 - Covering a branch means executing a path that contains the branch.
 - 100% branch coverage means selecting a set of paths such that each branch is included on some path.

BCPath 1	1-2-3(F)-10(F)-11-13
BCPath 2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
BCPath 3	1-2-3(T)-4(F)-10(F)-11-13
BCPath 4	1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13
BCPath 5	1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13

Table 4.5: Paths for branch coverage of the flow graph of Figure 4.7.

- Predicate coverage criterion
 - If all possible combinations of truth values of the conditions affecting a path have been explored under some tests, then we say that predicate coverage has been achieved.

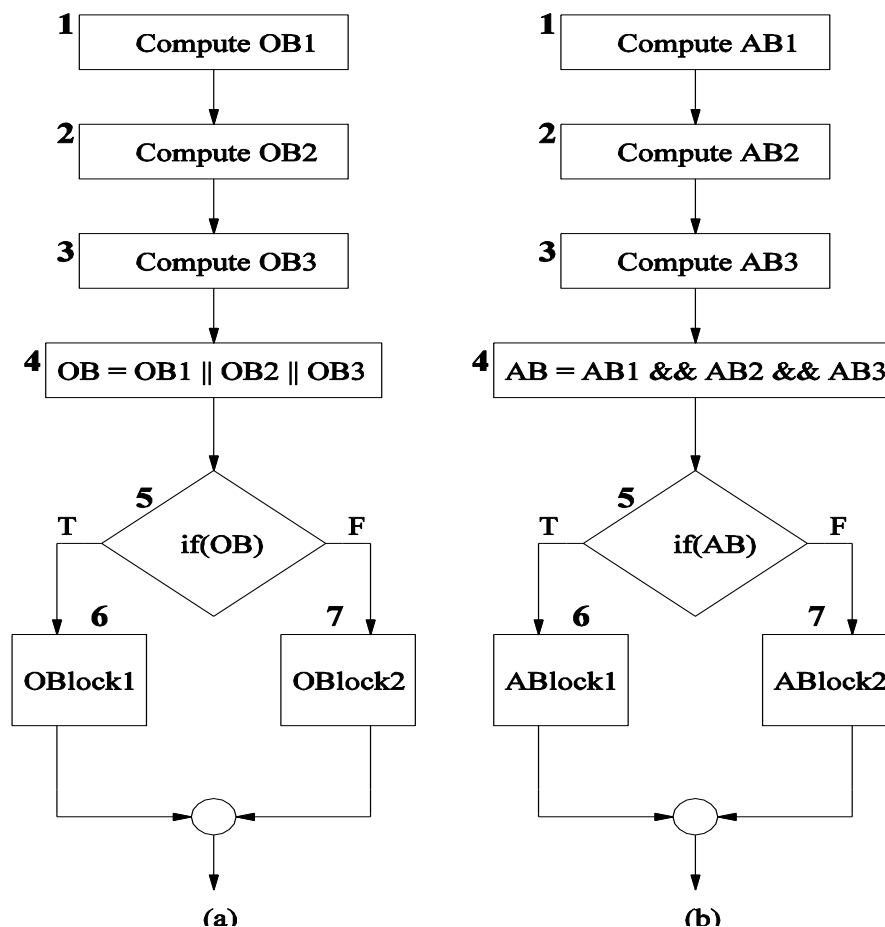


Figure 4.9: Partial control flow graph with (a) OR operation and (b) AND operation.

- Having identified a path, a key question is how to make the path execute, if possible.
 - Generate input data that satisfy all the conditions on the path.
- Key concepts in generating test input data
 - Input vector
 - Predicate
 - Path condition
 - Predicate interpretation
 - Path predicate expression
 - Generating test input from path predicate expression

- Input vector
 - An input vector is a collection of all data entities read by the routine whose values must be fixed prior to entering the routine.
 - Members of an input vector can be as follows.
 - Input arguments to the routine
 - Global variables and constants
 - Files
 - Contents of registers (in Assembly language programming)
 - Network connections
 - Timers
 - Example: An input vector for `openfiles()` consists of individual presence or absence of the files “file1,” “file2,” and “file3.”
 - Example: The input vector of `ReturnAverega()` shown in Figure 4.6 is `<value[], AS, MIN, MAX>`.

- Predicate
 - A predicate is a logical function evaluated at a decision point.
 - Example: $ti < AS$ is a predicate in node 3 of Figure 4.7.
 - Example: The construct **OB** is a predicate in node 5 in Figure 4.9.
- Path predicate
 - A path predicate is the set of predicates associated with a path.
 - **Figure 4.10:** An example path from Fig. 4.7:
 - 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.
 - **Figure 4.11:** The path predicate for the path shown in Figure 4.10.

$ti < AS$	\equiv True
$value[i] \neq -999$	\equiv True
$value[i] \geq MIN$	\equiv True
$value[i] \leq MAX$	\equiv True
$ti < AS$	\equiv False
$tv > 0$	\equiv True

- Predicate interpretation
 - A path predicate may contain local variables.
 - Example: $\langle i, t_i, t_v \rangle$ in Figure 4.11 are local variables.
 - Local variables play no role in selecting inputs that force a path to execute.
 - Local variables can be eliminated by a process called **symbolic execution**.
 - Predicate interpretation is defined as the process of
 - symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
 - A predicate may have different interpretations depending on how control reaches the predicate.

- Path predicate expression
 - An interpreted path predicate is called a path predicate expression.
 - A path predicate expression has the following attributes.
 - It is void of local variables.
 - It is a set of constraints in terms of the input vector, and, maybe, constants.
 - Path forcing inputs can be generated by solving the constraints.
 - If a path predicate expression has no solution, the path is infeasible.
 - **Figure 4.13:** Path predicate expression for the path shown in Figure 4.10.

$0 < AS \quad \equiv \text{True} \quad \dots\dots (1)$

$\text{value}[0] \neq -999 \quad \equiv \text{True} \quad \dots\dots (2)$

$\text{value}[0] \geq \text{MIN} \quad \equiv \text{True} \quad \dots\dots (3)$

$\text{value}[0] \leq \text{MAX} \quad \equiv \text{True} \quad \dots\dots (4)$

$1 < AS \quad \equiv \text{False} \quad \dots\dots (5)$

$1 > 0 \quad \equiv \text{True} \quad \dots\dots (6)$

- Path predicate expression
 - An example of infeasible path
 - **Figure 4.14:** Another example of path from Figure 4.7.
 - 1-2-3(T)-4(F)-10(T)-12-13
 - **Figure 4.15:** Path predicate expression for the path shown in Figure 4.14.

$0 < AS$	$\equiv \text{True} \dots\dots (1)$
$\text{value}[0] \neq -999$	$\equiv \text{True} \dots\dots (2)$
$0 > 0$	$\equiv \text{True} \dots\dots (3)$

- Generating input data from a path predicate expression
 - Consider the path predicate expression of Figure 4.13 (reproduced below.)

$$0 < AS \quad \equiv \text{True} \quad \text{..... (1)}$$

$$\text{value}[0] \neq -999 \quad \equiv \text{True} \quad \text{..... (2)}$$

$$\text{value}[0] \geq \text{MIN} \quad \equiv \text{True} \quad \text{..... (3)}$$

$$\text{value}[0] \leq \text{MAX} \quad \equiv \text{True} \quad \text{..... (4)}$$

$$1 < AS \quad \equiv \text{False} \quad \text{..... (5)}$$

$$1 > 0 \quad \equiv \text{True} \quad \text{..... (6)}$$

- One can solve the above equations to obtain the following test input data

$$AS = 1$$

$$\text{MIN} = 25$$

$$\text{MAX} = 35$$

$$\text{Value}[0] = 30$$

- Note: The above set is not unique.

- A program unit may contain a large number of paths.
 - Path selection becomes a problem. Some selected paths may be infeasible.
 - Apply a path selection strategy:
 - Select as many short paths as possible.
 - Choose longer paths.
 - There are efforts to write code with fewer/no infeasible paths.

- Control flow is a fundamental concept in program execution.
- A program path is an instance of execution of a program unit.
- Select a set of paths by considering path **selection criteria**.
 - Statement coverage
 - Branch coverage
 - Predicate coverage
 - All paths
- From source code, derive a CFG (compilers are modified for this.)
- Select paths from a CFG based on path selection criteria.
- Extract path predicates from each path.
- Solve the path predicate expression to generate test input data.
- There are two kinds of paths.
 - feasible
 - infeasible