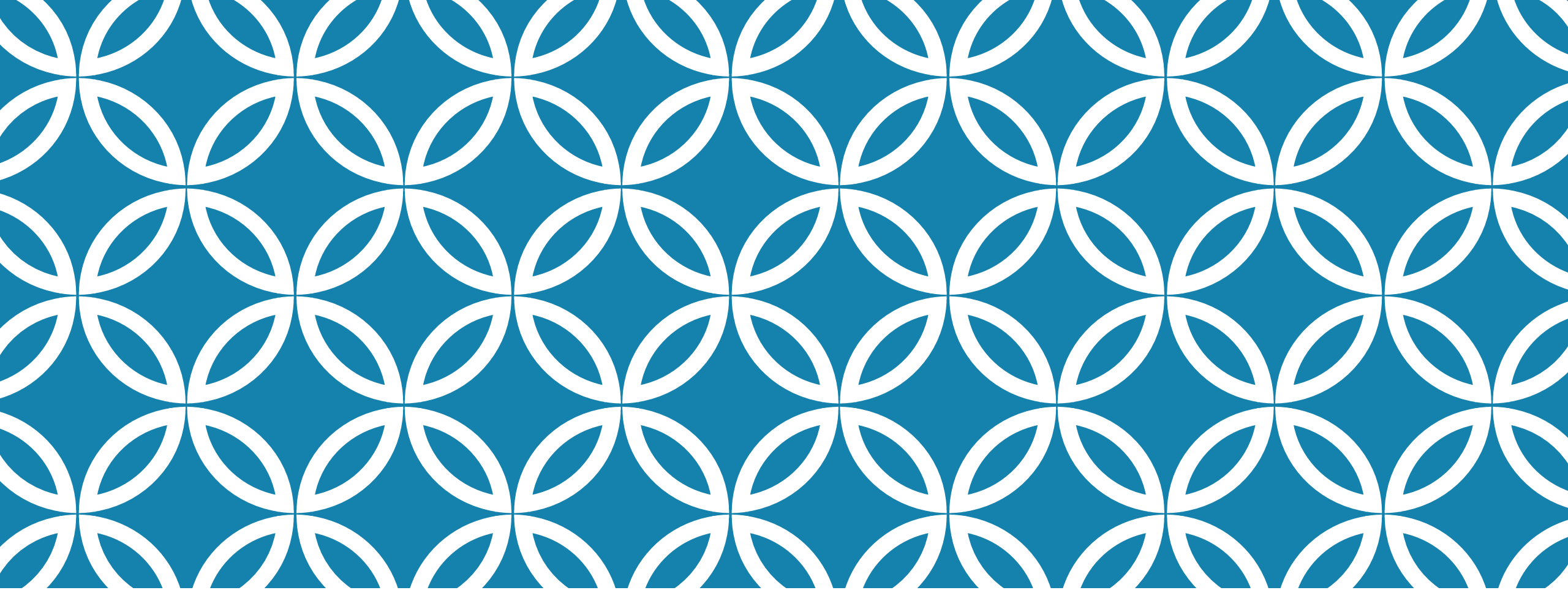


CHAPTER – 4
CONTROL FLOW TESTING

OUTLINE OF THE CHAPTER

- 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

BASIC IDEA

- 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.

OUTLINE OF CONTROL FLOW TESTING

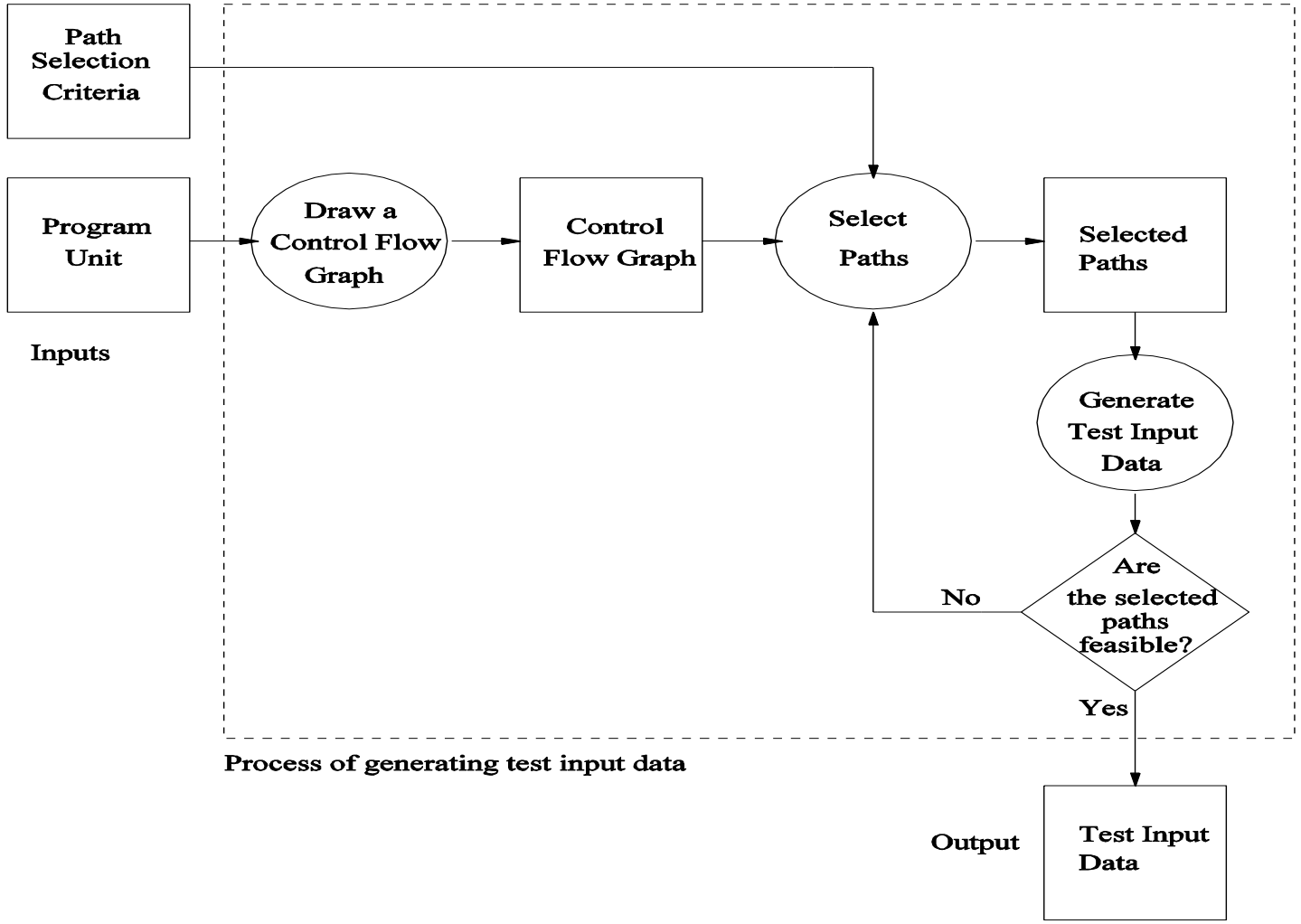


Figure 1: The process of generating test input data for control flow testing.

OUTLINE OF 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.

CONTROL FLOW GRAPH

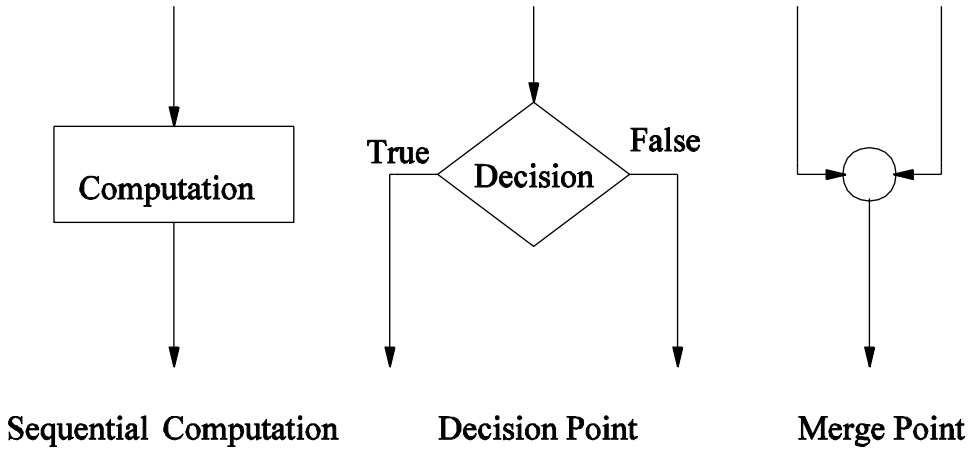


Figure 2: Symbols in a control flow graph

CONTROL FLOW GRAPH

```
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);
}
```

CONTROL FLOW GRAPH

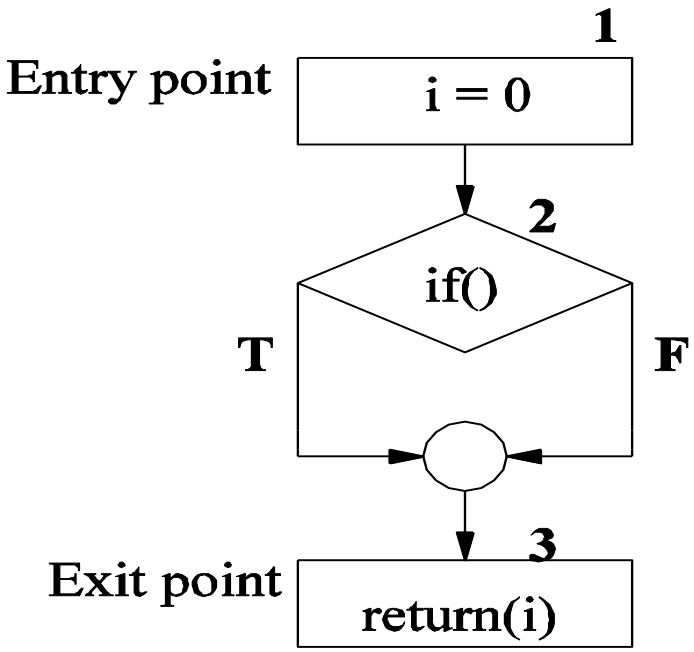


Figure 3: A high-level CFG representation of openfiles().

CONTROL FLOW GRAPH

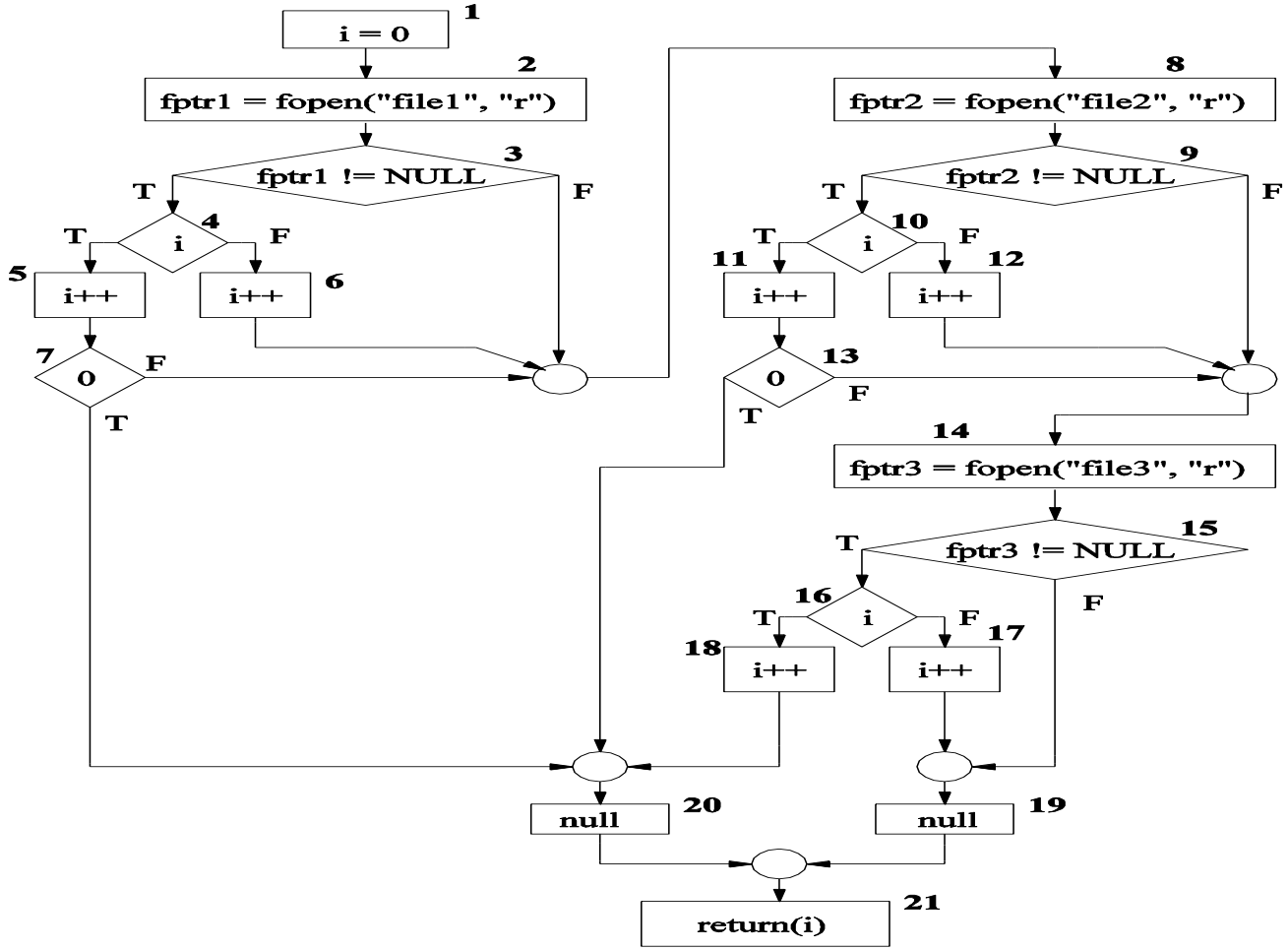


Figure 4: A detailed CFG representation of openfiles().

CONTROL FLOW GRAPH

```
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);
}
```

CONTROL FLOW GRAPH

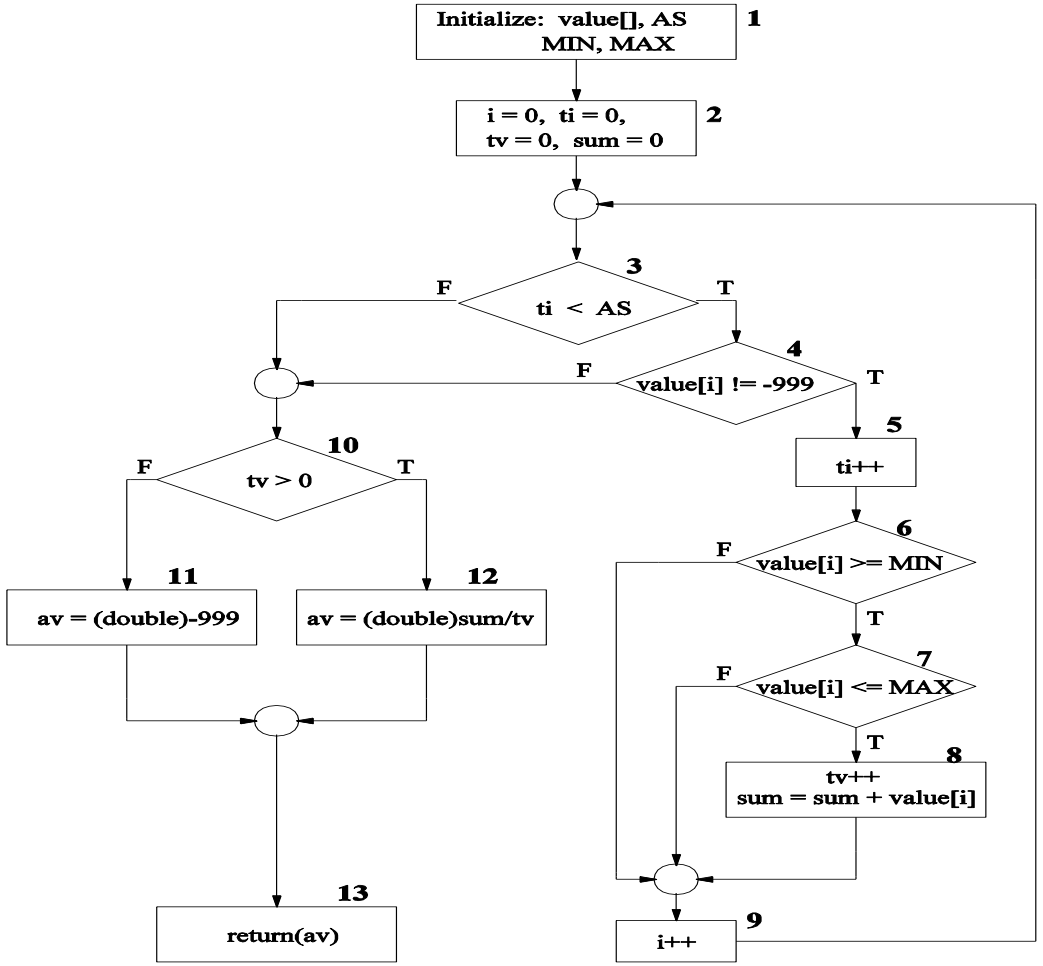


Figure 5: A CFG representation of ReturnAverage().

PATHS IN A CONTROL FLOW GRAPH

- A path is represented as a sequence of computation and decision nodes from the entry node to the exit node.
- A few paths in Figure 5
  - 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

PATH SELECTION CRITERIA

- 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.

PATH SELECTION CRITERIA

All-path coverage criterion: Select all the paths in the program unit under consideration.

- The openfiles() unit has 25+ paths.
- Selecting all the inputs will exercise all the program 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 1: Input Domain of openfiles()

PATH SELECTION CRITERIA

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 2: Inputs and paths in openfiles()

PATH SELECTION CRITERIA

- **Statement coverage criterion**
  - Statement here means node.
  - 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 3: Paths for statement coverage of the CFG of Figure 5.

PATH SELECTION CRITERIA

- **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.
  - Complete branch coverage means selecting a number of paths such that every branch is included in at least one path.

PATH SELECTION CRITERIA

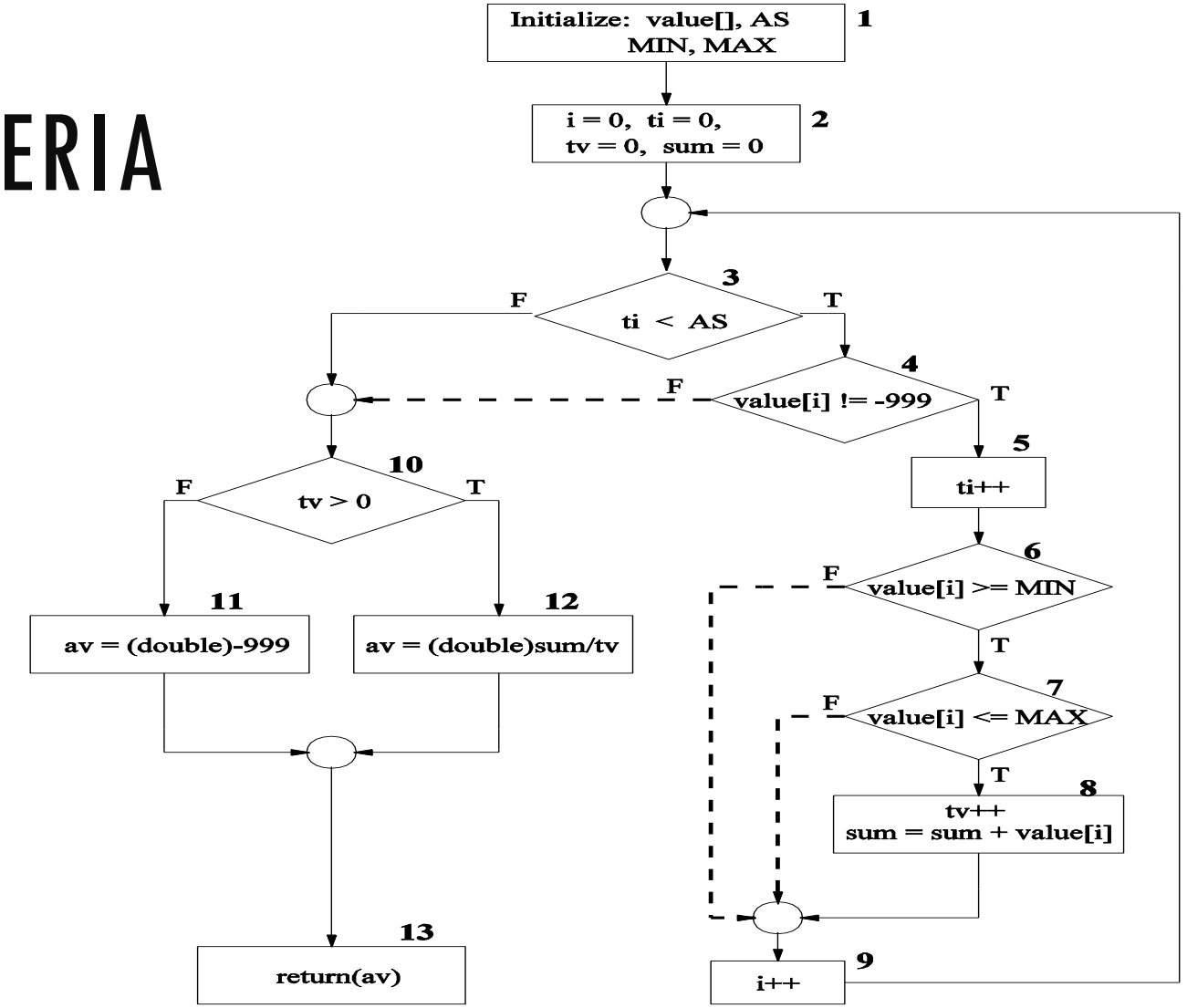


Figure 6: The dotted arrows represent the branches not covered by the statement covering in Table 3.

PATH SELECTION CRITERIA

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: Paths for branch coverage of the flow graph of Figure 5

PATH SELECTION CRITERIA

- **Predicate coverage/condition 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.

Cases	OB1	OB2	OB3	OB
1	T	F	F	T
2	F	F	F	F

Table 5: Two test cases providing complete statement coverage and branch coverage

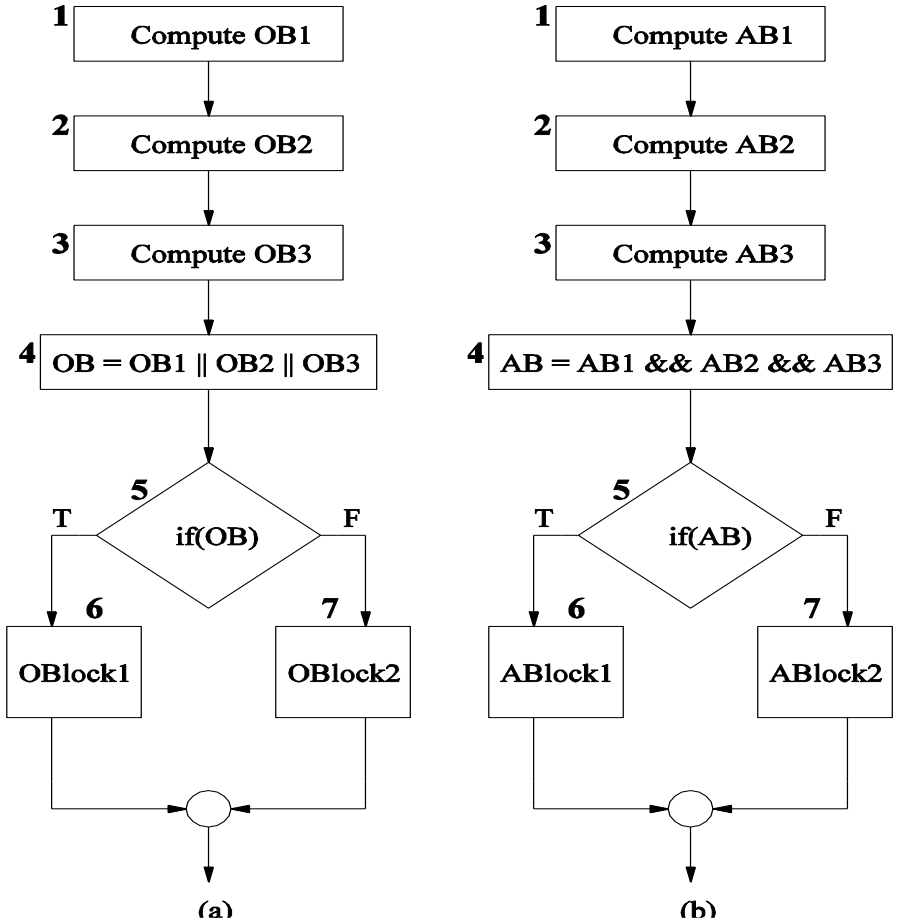


Figure 7: Partial control flow graph with (a) OR operation and (b) AND operation.

PATH SELECTION CRITERIA

- The False branch of node **5** is executed under exactly one condition, namely, when OB1 = False, OB2 = False, and OB3 = False, whereas the true branch executes under *seven* conditions.
- If all possible combinations of truth values of the conditions affecting a selected path have been explored under some tests, then we say that *predicate coverage* has been achieved.
- Therefore, the path taking the true branch of node **5** in Figure 7 must be executed for all seven possible combinations of truth values of OB1, OB2, and OB3, which results in OB = True.

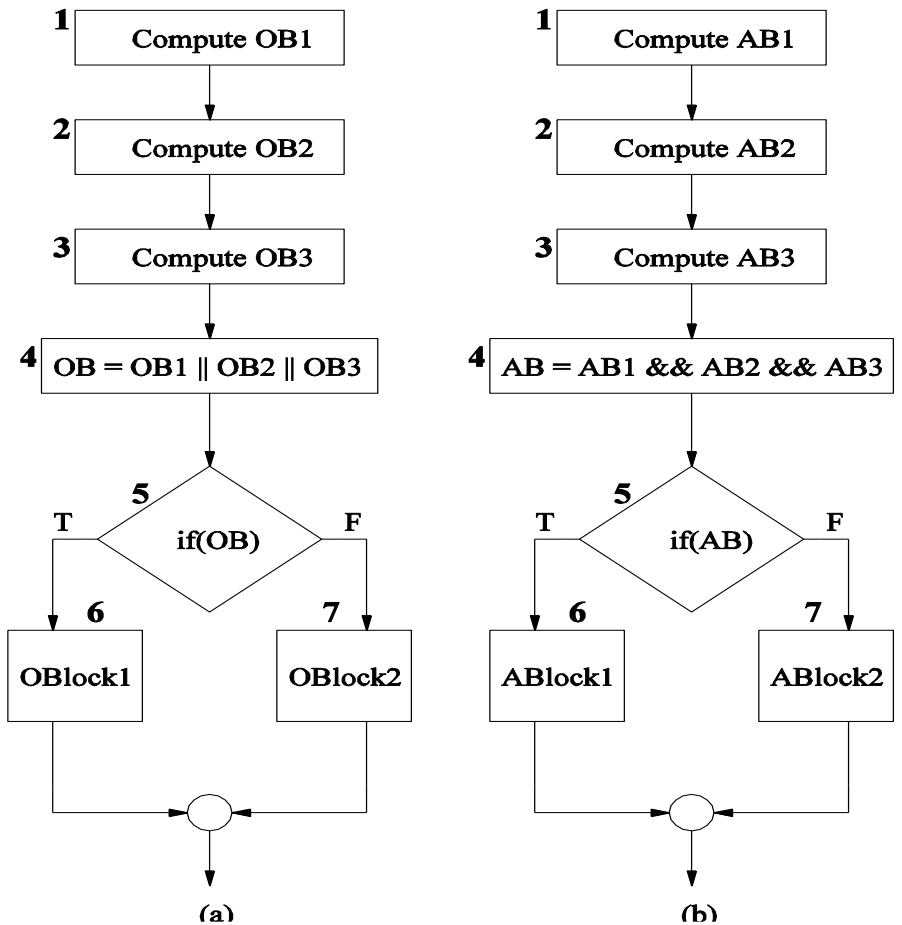


Figure 7: Partial control flow graph with (a) OR operation and (b) AND operation.

GENERATING TEST INPUT

- 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 predicate
  - Predicate interpretation
  - Path predicate expression
  - Generating test input from path predicate expression

GENERATING TEST INPUT

- 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 “files1,” “file2,” and “file3.”
  - Example: The input vector of ReturnAverege() shown in Figure 5 is <value[], AS, MIN, MAX>.

GENERATING TEST INPUT

- Predicate
  - A predicate is a logical function evaluated at a decision point.
  - Example: ti < AS is a predicate in node 3 of Figure 5.
  - Example: The construct OB is a predicate in node 5 in Figure 7.
- Path predicate
  - A path predicate is the set of predicates associated with a path.
  - An example path from Figure 5.
    - 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.
  - The path predicate for the path shown above.
    - ti < AS ≡ True
    - value[i] != -999 ≡ True
    - value[i] >= MIN ≡ True
    - value[i] <= MAX ≡ True
    - ti < AS ≡ False
    - tv > 0 ≡ True

GENERATING TEST INPUT

- Predicate interpretation
  - A path predicate may contain local variables.
  - Example: It is composed of elements of the input vector <value[], AS, MIN, MAX>, a vector of local variables <i, ti, tv>, and the constant −999.
  - The local variables are not visible outside a function but are used to hold intermediate results, pointer to array elements, and control loop iterations.
  - 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**.

SYMBOLIC SUBSTITUTION

```
public static int SymSub(int x1, int x2){
int y;
y = x2 + 7;
if (x1 + y >= 0)
     return (x2 + y);
else return (x2 - y);
}
```

- The input vector for the method in example is given by<x1, x2>.
- The method defines a local variable y and uses the constants 7 and 0.
- The predicate x1 + y >= 0 can be rewritten as x1 + x2 + 7 >= 0 by symbolically substituting y with x2 +7.
- The rewritten predicate x1 + x2 + 7 >= 0 has been expressed solely in terms of the input vector<x1,x2>and the constant vector<0,7>.
- 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.

GENERATING TEST INPUT

- 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.
  - Path predicate expression for the path → 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13 [Figure 5].
    - 0 < AS ≡ True …… (1)
    - value[0] != -999 ≡ True …… (2)
    - value[0] >= MIN ≡ True …… (3)
    - value[0] <= MAX ≡ True …… (4)
    - 1 < AS ≡ False …… (5)
    - 1 > 0 ≡ True …… (6)

GENERATING TEST INPUT

- Path → 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
- The rows here have been obtained from Side 24 [from path predicate] by combining each interpreted predicate in column 3 with its intended evaluation in column 1.

Interpretation of Path Predicate of Path in Figure 5

Node	Node Description	Interpreted Description
1	Input vector: < value[], AS, MIN, MAX >	
2	i = 0, ti = 0, tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(T)	value[i]! = − 999	value[0]! = − 999
5	ti++	ti = 0 + 1 = 1
6(T)	value[i] > = MIN	value[0] > = MIN
7(T)	value[i] < = MAX	value[0] < = MAX
8	tv++	tv = 0 + 1 = 1
	sum = sum + value[i]	sum = 0 + value[0] = value[0]
9	i++	i = 0 + 1 = 1
3(F)	ti < AS	1 < AS
10(T)	tv > 0	1 > 0
12	av = (double) sum/tv	av = (double) value[0]/1
13	return(av)	return(value[0])

Note: The bold entries in column 1 denote interpreted predicates.

GENERATING TEST INPUT

Path predicate expression

- An example of infeasible path
- Another example of path from Figure 5.
  - 1-2-3(T)-4(F)-10(T)-12-13
- Path predicate expression for the path shown above.

$0 < AS \equiv True \quad \ldots\ldots (1)$

$value[0] != -999 \equiv True \quad \ldots\ldots (2)$

$0 > 0 \equiv True \quad \ldots\ldots (3)$

Node	Node Description	Interpreted Description
1	Input vector: < value[], AS, MIN, MAX >	
2	i = 0, ti = 0, tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(F)	value[i]! = − 999	value[0]! = − 999
10(T)	tv > 0	0 > 0
12	av = (double)sum/tv	av = (double)value[0]/0
13	return(av)	return((double) value[0]/0)

Note: The bold entries in column 1 denote interpreted predicates.

GENERATING TEST INPUT

Generating input data from a path predicate expression

- Consider the path predicate expression (reproduced below.)

0 < AS	≡ True	…… (1)
value[0] != -999	≡ True	…… (2)
value[0] >= MIN	≡ True	…… (3)
value[0] <= MAX	≡ True	…… (4)
1 < AS	≡ False	…… (5)
1 > 0	≡ True	…… (6)

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

AS	= 1
MIN	= 25
MAX	= 35
Value[0]	= 30

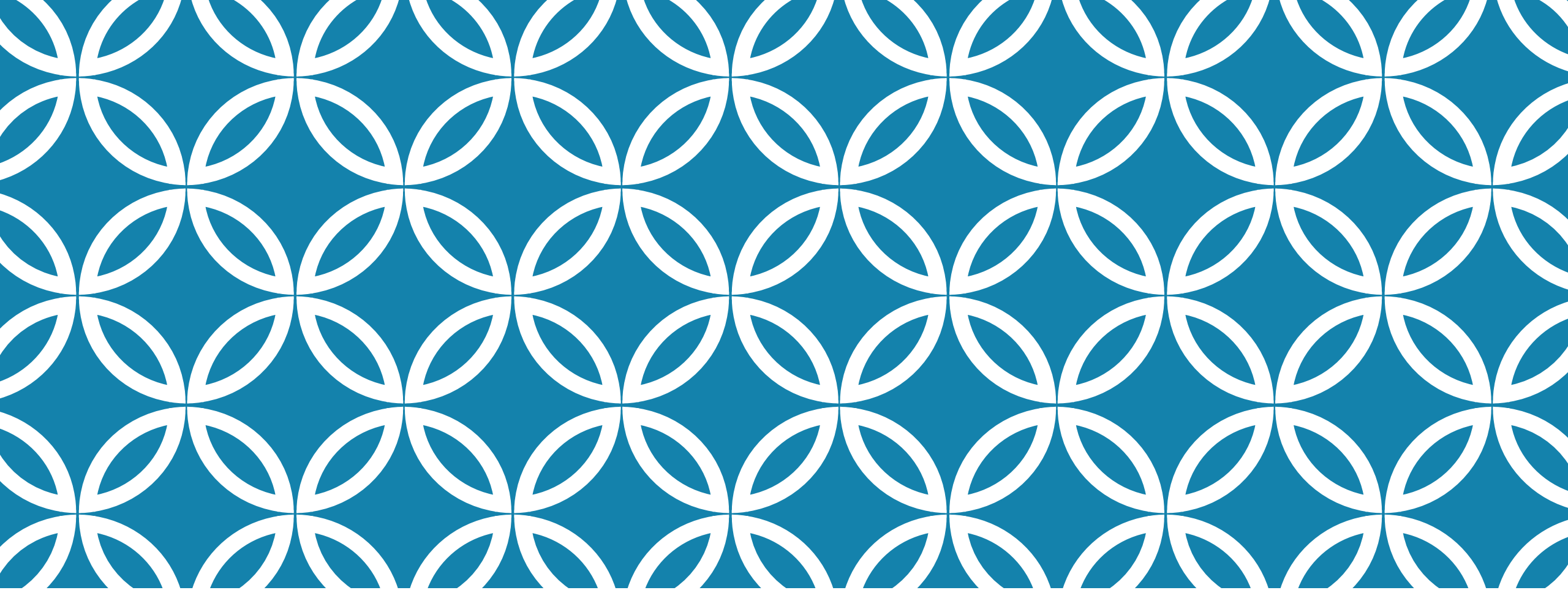
- Note: The above set is not unique.

CONTAINING INFEASIBLE PATHS

- 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.

SUMMARY

- 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



THANK YOU!!