7-5-1995

Weakest Pre-Condition and Data Flow Testing

Griffin David McClellan
Portland State University

10.15760/etd.7076

THESIS APPROVAL

The abstract and thesis of Griffin David McClellan for the Master of Science degree in Computer Science were presented July 5, 1995 and accepted by the thesis committee and the department.

COMMITTEE APPROVALS: [redacted],
Sergio Antoy, Advisor

[redacted],
Dick Hamlet

[redacted],
Jim Hein

[redacted],
Dorothy Williams, Representative of the Office Of Graduate Studies

DEPARTMENT APPROVAL: [redacted],
Warren Harrison, Chair
Department of Computer Science

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted], on 14 September 1995

Abstract

An abstract of the thesis of Griffin David McClellan for the Master of Science in Computer Science presented July 5, 1995.

Title: Weakest Pre-Condition and Data Flow Testing

Current data flow testing criteria cannot be applied to test array elements for two reasons:

1. The criteria are defined in terms of graph theory which is insufficiently expressive to investigate array elements.

2. Identifying input data which test a specified array element is an unsolvable problem.

We solve the first problem by redefining the criteria without graph theory. We address the second problem with the invention of the *wp_du method*, which is based on Dijkstra's weakest pre-condition formalism. This method accomplishes the following: Given a program, a def-use pair and a variable (which can be an array element), the method computes a logical expression which characterizes all the input data which test that def-use pair with respect to that variable. Further, for any data flow criterion, this method can be used to construct a logical expression which characterizes all test sets which satisfy that data flow criterion. Although the wp_du method cannot avoid unsolvability, it does confine the presence of unsolvability to the final step in constructing a test set.

WEAKEST PRE-CONDITION AND DATA FLOW TESTING

by
GRIFFIN DAVID MCCLELLAN

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1995

Contents

Introduction

This thesis solves a problem in software testing by applying a method of formal verification of programs. We assume a modest familiarity with:

- the syntax and proof methods of predicate logic [Gries81]
- formal verification of programs, including Hantler and King's symbolic execution [HanKin76] and Dijkstra's weakest pre-condition formalism [Dijkstra76]
- the mathematical concepts of partitions and equivalence classes [Durbin92]
- theory of computation, specifically unsolvability [ManGhe87]
- graph theory [ManGhe87]
- finite state machines [ManGhe87]
- control flow and data flow testing [Hamlet88], [RaWey85], [FraWey88]
- programming in an imperative, structured language
- flow charts

Current data flow testing criteria cannot be applied to test array elements for two reasons:

1. The criteria are defined in terms of graph theory which is insufficiently expressive to investigate array elements.

2. Identifying input data which test a specified array element is an unsolvable problem.

We solve the first problem by redefining the criteria without graph theory. We address the second problem with the invention of the *wp_du method*, which is based on Dijkstra's weakest pre-condition formalism. This method accomplishes the following: Given a program, a def-use pair and a variable (which can be an array element), the method computes a logical expression which characterizes all the input data which test that def-use pair with respect to that variable. Further, for any data flow criterion, this method can be used to construct a logical expression which characterizes all test sets which satisfy that data flow criterion. The achievement of the wp_du method is that it reduces the unsolvable problem of identifying input data which test a specific array element to the problem of generating solutions to a logical expression.

The first three chapters provide the background and motivation for the problem solved in this work. These chapters review many of the topics enumerated above. The fourth chapter provides a framework for the solution, and the remaining chapters explain and demonstrate the solution. The solution presented here is the *wp_du method*, where "wp_du" is pronounced by saying the name of each letter.

For simplicity, the programming examples will be done in the following subset of Pascal containing:

- variables of any simple type (integer, real, character, boolean, real, or enumerated) and arrays with simple base types.
- assignment statements and readln and writeln statements
- if statements
- while statements
- functions where actual parameters are passed by value

Unless otherwise specified, all variables are integers. Further, for reasons that will become apparent in chapter 5, we also reserve the identifiers `status`, `defined`, `not_defined`, `def_used`, and `I` for our own use.

1 Fundamentals of Path Testing Theory

This chapter reviews the background in testing theory which is required to appreciate the problem addressed by this thesis.

More specifically, this chapter includes sections which discuss:

- the use of software testing to demonstrate the correctness of a program,
- partition testing, which is a general testing paradigm,
- path testing, which is a kind of partition testing,

Correctness of a Program and Software Testing

Given a program and some kind of description of how that program is supposed to behave, how can we demonstrate that the program satisfies that description? This is the correctness problem.

For simplicity, we will interpret any program as behaving like a mathematical function. That is, when we run the program, we supply it with an input datum from the set of all possible input data. An input datum may be a number, or a sequence of numbers, or something vastly more complicated. The program uses this input datum to compute a result (if the program terminates). (The cautious reader is assured that a result in mathematical logic [ManGhe87] justifies interpreting any program and its input in this manner.) We will assume that, if the program terminates, the description we have of the program's intended behavior allows us to identify any incorrect computations.

The simplest solution to the correctness problem is to test the program with every possible input and check for the correct results. However, even if we overlook the problem of non-terminating programs, the number of possible inputs to any program executable on a contemporary computer is so vast that there are very few such programs we could check in this manner before the time at which our sun is expected to burn out.

A more sophisticated solution to the correctness problem is to prove the program is correct using some sort of program verification calculus. Although many such calculi exist [Hoare69] [Dijkstra72] [HanKin76], they are not often used in practice, because the application of such a calculus is too difficult and time-consuming. Indeed, proving a program correct with respect to a description of that program usually takes much longer than designing and implementing the program, even if a computer helps manage the proof.

Testing is another attempt at solving the correctness problem, and without question, it is the most commonly used method for demonstrating that a program does what it was intended to do. The goal of testing is to construct a set of input data, called the *test set*, which is a subset of the set of all possible input data. The ideal test set would have the property that executing the program with each test datum will reveal all the errors in the program.

Conversely, if the test set is executed without any errors, then we can conclude that the program will behave in accordance with its description for all input data. Of course, the test set must be small enough that the time required to execute the program once for each test datum is not prohibitive.

Unfortunately, testing cannot solve the correctness problem. More specifically, there is no algorithm which constructs, for an arbitrary program, the ideal test set described above. The reason for this is expressed in Dijkstra's pithy observation: testing can reveal the presence of errors, but not their absence [Dijkstra72]. This insight is made precise in a proof by Howden involving recursive function theory [Howden76]. In spite of this result, we study testing because, at this time, it appears to be the only practical way to reveal errors and to increase our feeling that a program behaves as we intend it to.

Due to the limitation explained in the previous paragraph, researchers in testing often do not develop algorithms for generating test sets for programs, but instead develop specifications of what it means to adequately test a program. Such a specification is called a *testing criterion.* There are many different testing criteria, each with their own prescriptions for what constitutes an acceptable test of a program. In this thesis, we will encounter many different testing criteria. If a test set tests all the features which a testing criterion prescribes, then we say that *the test set satisfies that testing criterion for that program.*

A testing criterion may have the following two limitations: First, the criterion may not indicate how to construct a test set which satisfies that criterion for a given program. Second, given a program and a testing criterion, we may not be able to decide in a finite amount of time if a particular test set satisfies the criterion for that program or even if such a test exists at all. We will explore these issues later in this chapter.

In summary, although testing cannot be used to solve the correctness problem, we will nonetheless explore methods for constructing test sets which we hope will often uncover errors.

Partition Testing

Most of the testing criteria which have been proposed are based on the idea of partition testing, which divides the set of input data into equivalence classes, and then constructs a test set by randomly selecting elements from each equivalence class. A crucial issue in partition testing is the construction of the equivalence classes. As Hamlet and Taylor observe [HamTay90]:

> A partition can be defined using all the information about a program. It can be based on requirements or specifications (one form of "blackbox" testing), on features of the code ("structural" testing), even on the process by which the software was developed, or on the suspicions and fears of a programmer.

Although these authors have shown that partition testing is not as effective as our intuition suggests, we will assume that partition testing is worth pursuing.

Two final comments: First, the equivalence classes produced in partition testing sometimes contain common elements and therefore are not partitions in the strict mathematical sense. This is not usually a problem. Second, just as in abstract algebra, we use predicate logic to describe the equivalence classes. Examples follow in the next section.

Path Testing

We review a kind of partition testing called *path testing.* This section has four parts:

1) the intuitive idea of a path,
2) the graph theoretic definition of path,
3) a short discussion of three simple path testing criteria.
4) the use of predicate logic to characterize the input data which exercise specific paths,

The Intuitive Idea of a Path

Imagine a program is executed with an input datum. During the execution of the program, certain statements in the program are executed in a specific order. Which statements are executed, and in what order, is determined by the program, the input datum, and the semantics of the language in which the program is written. Provisionally, we shall say that a path is the list of the statements that were executed, for some input datum, listed in the order in which they were executed. If an input datum causes a program to execute the statements in the path, in the order they appear in the path, we say that the input datum exercises that path.

A number of subtle points must be addressed.

First, two statements which are identical in appearance, but occur in different parts of the same program, are considered to be different statements. Thus, a statement is identified, not just by its syntactic form and semantic content, but by its position in the program.

Second, if a statement is inside the body of a loop, it may be executed many times during the execution of the program. In this case, the same statement will appear in the path as many times as it was executed.

Third, most programming languages define the syntactic form of statements recursively. The result is that some statements contain other statements. (An example of this is the block statement in Pascal, which is composed of a sequence of statements bracketed by the keywords **begin** and **end**.) Therefore, we distinguish between two kinds of statements: simple statements, which do not contain other statements, and compound statements, which contain other statements.

A more precise definition of path is that it is the list of the simple statements that were executed, for a given input datum, listed in the order they were executed. When a compound statement is executed, we represent it in a path by listing the the simple statements within that compound statement, in the order in which they were executed. Typically, the boolean expressions which control **if** and **while** statements are not included in the path, because their inclusion is redundant. (The boolean expressions are redundant because they can be inferred from the list of simple statements which are executed.)

Finally, notice that for many programs, different input data will cause the same statements to be executed in the same order. We say that these input data exercise the same path.

The Graph Theoretic Definition of Path

The notion of a path has been formalized using graph theory. In this section, we display this formalization and investigate a problem created by this definition of path.

The first step in representing a path using graph theory is to represent the program itself as a graph. Such a representation is called a *program graph* (or *flow graph*). A program graph is a flow chart with the following differences:

1. All the different shapes that are used in a flow chart are replaced by circles called *nodes*. Thus, the diamond which usually represents a decision is replaced by a circle, as is the box that usually represents an assignment statement. The arrows which connect the nodes are called *edges*. A program graph is composed entirely of nodes and edges.

2. For the purposes of identification, every node has a unique number. Each node is either empty or contains just one simple statement. An empty node is a node which is not associated with any simple statement. Empty nodes correspond to the beginning of **if** or **while** statements in the program.

3. The boolean expressions which control **if** and **while** statements do not appear in nodes. Rather, they label the edges which depart from the empty node which represents the beginning of the **if** or **while** statement.

Three notes: First, we don't represent variable declarations in the program graph. Second, some authors consolidate any textually contiguous group of simple statements into a single node. In the interest of simplicity, we do not follow that approach. Lastly, Rapps and Weyuker offer a formal definition of a program graph for an unstructured language [RaWey85].

For example, consider the following program fragment, where all the program variables are integers and `odd` is a boolean function with its standard meaning.

```
if x < 0 then
     y := 1
else
     y := 2;

if odd( x ) then
     z := 1
else
     z := 3;

a := y + z;
```

Figure 1.1

Its corresponding program graph is:

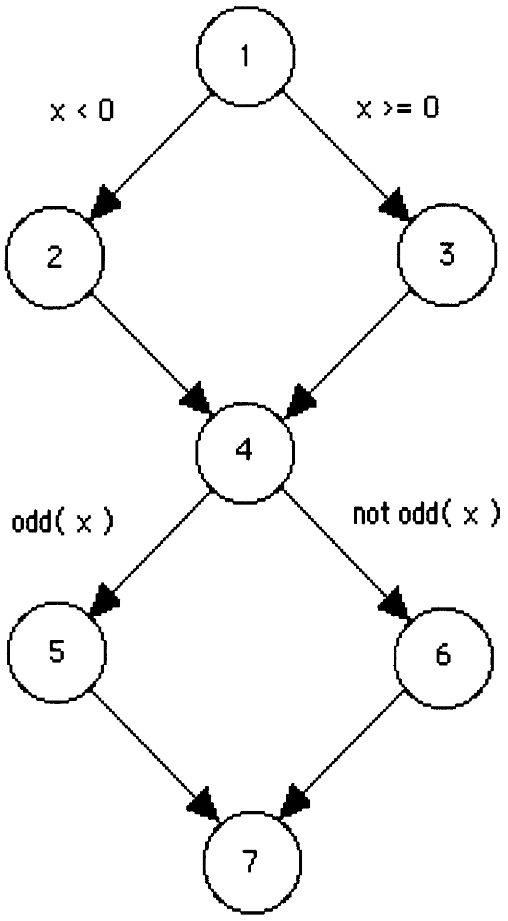

Figure 1.2

Notice that the boolean test of each **if** statement and its negation are associated with edges which lead to the then and else clauses of the **if** statement, respectively. The correspondence between simple statements in the example program and the nodes of the program graph is as follows:

node	statement
1	`empty` (the beginning of the first **if** statement)
2	`y := 1;`
3	`y := 2;`
4	`empty` (the beginning of the second **if** statement)
5	`z := 1;`
6	`z := 3;`
7	`a := y + z;`

Figure 1.3

It should be intuitively clear how to construct a program graph from an arbitrary program.

Using the graph theoretic formalism, a path through the program is defined to be a sequence of nodes which are connected by edges (respecting the directions of the arrows). In this work, *path* will have the aforementioned definition. A complete path begins with the node associated with the first statement in the program and ends with the node associated with the last statement in the program. The set of complete paths for the above program is `( 1, 2, 4, 5, 7 )`, `( 1, 2, 4, 6, 7 )`, `( 1, 3, 4, 5, 7 )`, and `( 1, 3, 4, 6, 7 )`. A path which is not complete, but nonetheless acceptable is `( 2, 4, 5 )`. Some sequences of nodes which are not paths are `( 7, 6, 4, 3, 1 )`, `( 5, 4, 2 )`, and `( 5, 4, 6 )` because they do not respect the arrows. If a path is a sub-sequence of another path, we say the latter contains the former. For example, `( 1, 2, 4, 5, 7 )` contains `( 2, 4, 5 )`.

Three Simple Path Testing Criteria

Given the formalization of the idea of a path, let's examine three path testing criteria. Recall that a testing criteria is a specification of what features of a program should be tested. More specifically, a path testing criterion specifies what kind of paths need to be exercised for the program to be adequately tested. Often, for a given path testing criterion and program, there are many sets of paths which satisfy that criterion for that program.

Once we have identified a set of paths which satisfies our chosen criterion, we need to construct a test set with the property that: after the program has been run with each test datum, all the paths in our set of paths have been exercised. Such a test set satisfies that criterion for that program. We address this issue of identifying the input data which exercise a given path in the next section.

Statement and branch coverage were two of the first widely-used testing criterion. When applying the former, we require that our test set contain test data such that after we have executed the program with each test datum, all the statements in the program will have been executed at least once. When applying branch coverage, we require that our test set cause each boolean expression in an **if** or **while** statement to evaluate to `true` at least once and `false` at least once.

Although it may not be immediately obvious, both statement and branch testing can be regarded as path testing criterion. Statement coverage requires that, from the set of all possible paths through a program, we choose a set of paths such that each node in the program graph appears at least once in our set. (For this reason, statement coverage is also called *all nodes* testing.) Branch coverage requires that we choose a set of paths such that every edge appears at least once in our set. (Branch coverage is also called *all edges* testing.)

Let's apply these criteria to the following code fragment and its program graph.

```
readln( x );

while not is_prime( x ) do begin
     writeln( x );
     x := 2 * x - 1
end;

writeln( 'done' );
```

Figure 1.4

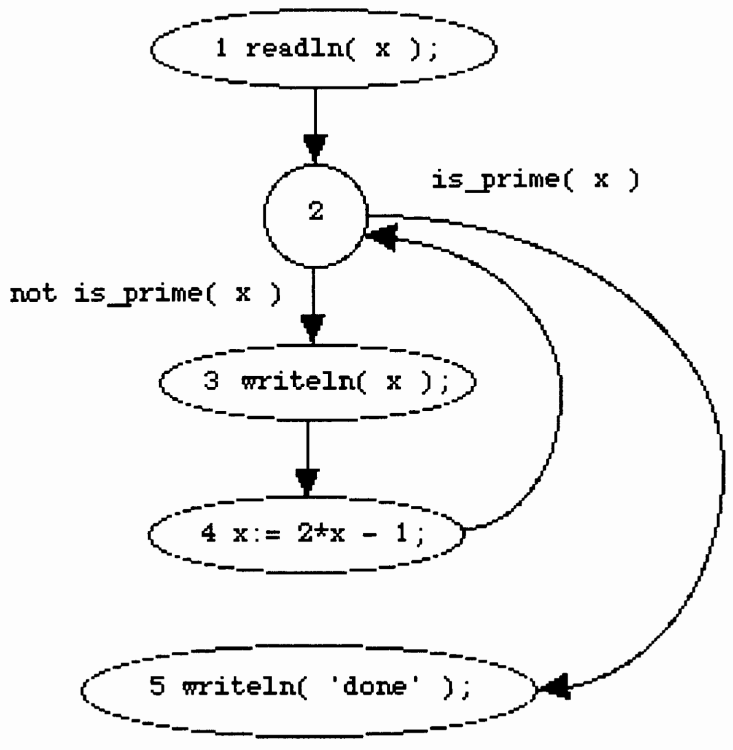

Figure 1.5

Regarding statement coverage, notice that there are many sets of paths we can choose which include all the nodes. `{ ( 1, 2, 5 ), ( 1, 2, 3, 4, 2, 3, 4, 2, 5 ) }` is such a set. The singleton set containing just `( 1, 2, 3, 4, 2, 5 )` is another. The reader should see that the latter is the smallest set of paths which satisfies the statement coverage criterion.

Similarly, there are many sets of paths which include all the edges. `{ ( 1, 2, 5 ), ( 1, 2, 3, 4, 2, 5 ) }` is one of many sets of paths which satisfy the branch coverage criterion.

Although statement and branch coverage are widely used, it is well known that they miss many common errors [FraWey88]. These weaknesses motivated the search for more perceptive testing criteria.

The final testing criterion we discuss here is all-paths testing. All-paths testing is often discussed, but never used. In understanding why it is never used, we will appreciate a subtle problem with our graph theoretic formalization of the idea of a path.

The **all-paths** testing criterion requires that a test set cause every path in the program to be exercised at least once. The problem with all-paths testing is that, for any program with a loop, there are an infinite number of paths through that program, and exercising an infinite number of paths, would require an infinite test set, which is of little use to finite beings.

To see why any program with a loop contains an infinite number of paths, return to the previous program fragment. All of the following are possible paths through the program: `( 1, 2, 5 )`, `( 1, 2, 3, 4, 2, 5 )`, `( 1, 2, 3, 4, 2, 3, 4, 2, 5 )`, and `( 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 5 )`. Indeed, any path which begins with node 1, contains any number of instances of `( 2, 3, 4 )` as contiguous sub-paths, and ends with the sub-path `( 2, 5 )` is an acceptable path by our graph theoretic definition.

Recall that our definition of path is any sequence of nodes which is consistent with the arrows in the program graph. The definition does not require that the path correspond to some possible execution of the program arising from some input datum. In other words, the definition of graph does not require us to pay attention to the boolean conditions which annotate some of the edges. For now, we just take note of the problem. We address the implications of the problem in detail in the chapter on data flow testing. Now we turn to the problem of describing the input data which exercise a path or a set of paths.

Predicate Logic Characterizes the Input Data Which Exercise a Path

In the previous section, we reviewed three simple path testing criteria. Each of those criteria required that we identify a set of paths with some property. Once we had the set of paths, we then needed to construct a test set which exercises those paths. We discuss this topic in the section on symbolic execution in Chapter three. For now, we just want to explore how, given a path, we can describe the input data which exercise that path.

For every path, there is a corresponding set of input data which exercises this path. We can use the logical expressions of predicate logic to characterize the input data which exercises that path. The following table illustrates the paths through the the program in Figure 1.1 and the logical expressions which characterize the input data which exercise those paths.

path	logical expression
`( 1, 2, 4, 5, 7 )`	`x < 0 and odd( x )`
`( 1, 2, 4, 6, 7 )`	`x < 0 and not odd( x )`
`( 1, 3, 4, 5, 7 )`	`x >= 0 and odd( x )`
`( 1, 3, 4, 6, 7 )`	`x >= 0 and not odd( x )`

In the chapter discussing symbolic execution, we will consider how such logical expressions can be constructed.

Finally, notice that, for any path, there exists an equivalence class of input data which causes that path to be exercised [Gries81]. We describe that class using predicate logic. Further, if we specify a set of paths, we have also specified a set of equivalence classes of input data. Each equivalence class contains the input data which causes one of the paths to execute. For example, if we specify the two paths `( 1, 2, 4, 5, 7 )` and `( 1, 3, 4, 6, 7 )` then the set of input data which will cause one of these two paths to be traversed is characterized by the predicate `( x < 0 and odd( x )) or ( x >= 0 and not odd( x ))`. This observation allows us to reason about paths and later translate our reasoning about paths back to classes of input data.

This concludes our overview of the fundamentals of path testing. In the next section, we explore a particular kind of path testing, data flow testing, which contains a problem which motivates this thesis.

2 Data Flow Testing

In the last chapter, we discussed three simple path testing criteria. Each of these criteria have disadvantages: statement and branch coverage miss common programming errors and all-paths testing, while it detects many errors, is often impossible to apply. Researchers in testing theory searched for a middle ground between statement and branch testing and all-paths testing. One family of criteria discovered in this middle ground was data flow testing, which is a type of path testing.

This section has five parts which discuss:
1) the basic idea of data flow testing,
2) three data flow testing criteria,
3) the problem of infeasible paths,
4) feasible data flow testing criteria,
5) unsolvability and feasible data flow criteria

Before we begin, note that the data flow criteria we define in this chapter do not allow us to test individual array elements. In Chapter four, we will extend these criteria to overcome this limitation. Finally, in the interest of clarity and simplicity, our definitions of the data flow criteria and some of its supporting vocabulary are slightly different than the original definitions. All the concepts remain the same.

The Basic Idea of Data Flow Testing

Historically, the first path testing criteria in common use were called *control flow* criteria, meaning they examined the branch and loop structure of the program. Statement and branch coverage are both control flow criteria. However, there is more to a program than its control structure, for example, the manner in which variables are defined and used. This was the insight of Rapps and Weyuker in their seminal paper *Selecting Software Test Data Using Data Flow Information* [RaWey85] where they introduced data flow testing.

There is no better motivation for data flow testing than the following [FraWey88]:

> These [data flow] criteria are based on the intuition that one should not feel confident that a variable has been assigned the correct value at some point in the program if no test data cause the execution of a path from the assignment to a point where the variable's value is subsequently used.

Thus, the basic idea of data flow testing is that we want to exercise, for each variable in the program, at least some of the various paths which cause that variable to be assigned a value and then used.

To clarify this intuition, let's consider the following example which computes x mod y for non-negative x and positive y. (Note: there is no error in this program.)

```
readln( x, y );

if ( x >= 0 ) and ( y > 0 ) then begin
     while x >= y do
          x := x - y;

     writeln( x )
end else
     writeln( 'error' );
```

Figure 2.1

The corresponding program graph is displayed below. Note that to facilitate reading the program graph, the simple statements in the program are displayed within their corresponding nodes.

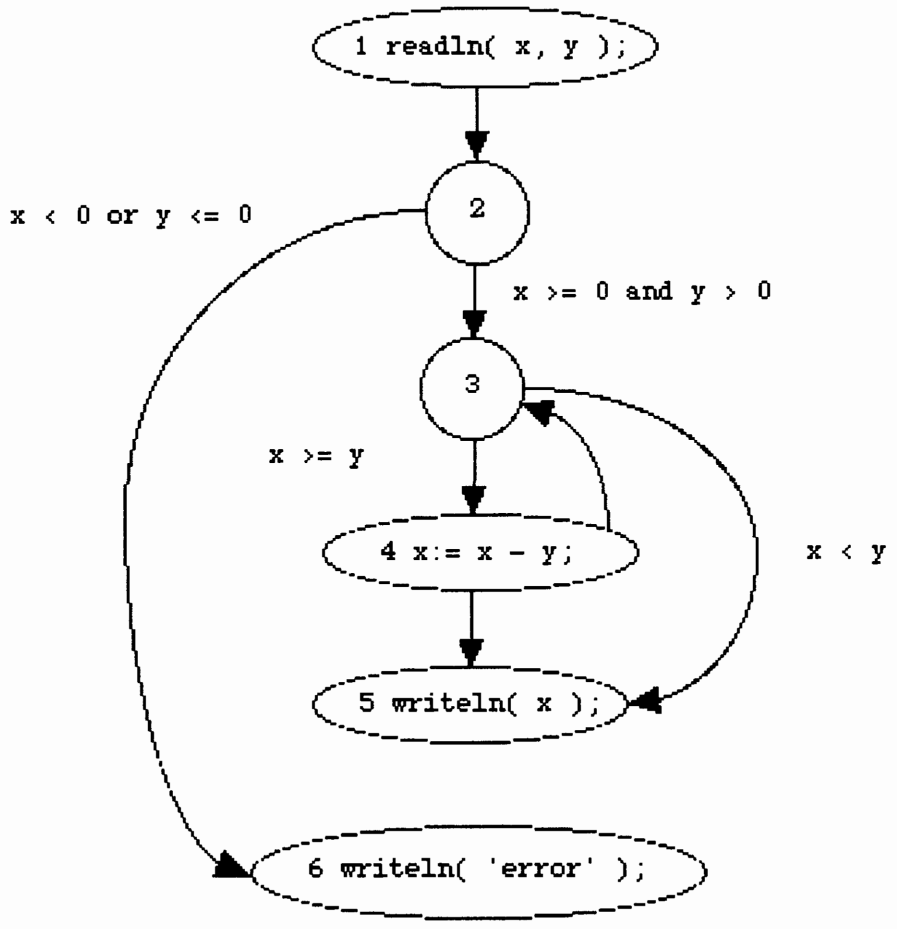

Figure 2.2

Before we continue with this example, note that data flow testing borrows some terminology from code optimization. In particular, the phrase "a variable is defined" means that the variable is assigned a value and to say that "a variable is used" is to say that the value of the variable is accessed.

To clarify how data flow testing works, let's identify all the possible definitions and uses of the variables `x` and `y`. This identification will be facilitated by the introduction of a notation for naming places in the program graph where a variable is defined or used.

Note that variables are always defined in simple statements (either assignment statements or readln statements). We will represent a statement of definition with the symbol "`DEF:`", followed by the variable being defined, a comma, and the node number. Thus, the `readln( x, y );` can be represented as either `DEF: x, 1` or `DEF: y, 1`, depending on which variable we want to focus on.

When we turn to identifying the places in a program graph where a variable is used, we see that variables are used in two different ways: within nodes or on the edges between nodes. In terms of the program itself, this means that a variable is either used in a simple statement or in a boolean expression controlling an **if** or a **while** statement. Examples of the former are variables appearing in the right hand side of an assignment statement or in an output statement. Such examples are called *c-uses* of the variable, which stands for "computation" use. Boolean expressions which control **if** or **while** statements are called *p-uses* of the variables within the boolean expression, which stands for "predicate" use.

C-uses are represented with the symbol "`USE:`", followed by the variable being used, a comma, and the node number. The `writeln( x );` statement can be represented by `USE: x, 5.`

Identifying a p-use with a part of the program graph requires some care. In data flow testing, each edge labelled with a boolean expression is a p-use for each of the variables occurring in that expression. Recall that each boolean expression which controls an **if** or **while** statement is represented by two edges in the program graph; one of the edges corresponds to the case where the expression evaluates to `true` and the other edge corresponds to the case where the expression evaluates to `false`. Therefore, each variable which occurs in a given boolean expression which controls an **if** or **while** statement has two p-uses in that expression. For example, the p-use of `x` which occurs in the **while** statement is identified with the edges `( 3, 4 )` and `( 3, 5 )`. These edges are also p-uses of `y`.

The reason each p-use is identified with two edges is because the inventors of data flow testing wanted to create a criterion which was at least as thorough as branch coverage. If each p-use is identified with both edges, their hope was that when we test all p-uses of all variables, we also test all branches in the program.

We will identify p-uses with the symbol "`USE:`", followed by the variable being used, a comma, and the appropriate edge, represented as an ordered pair of nodes which surround the edge.

Finally, note that when we mention "a *definition* of a variable in a program graph," we mean the node which corresponds to a statement where that variable is defined. When we mention "a *use* of a variable in a program graph," we mean a node or edge which corresponds to a statement where that variable is used.

The following is a list of all the possible ways in which variable `x` can be defined and used:

```
DEF: x, 1 USE: x, 4
DEF: x, 1 USE: x, 5
DEF: x, 4 USE: x, 4
DEF: x, 4 USE: x, 5
DEF: x, 1 USE: x, (2, 3)
DEF: x, 1 USE: x, (2, 6)
DEF: x, 4 USE: x, (3, 4)
DEF: x, 4 USE: x, (3, 5)
```

The possible ways in which variable `y` can be defined and used:

```
DEF: y, 1 USE: y, 4
DEF: y, 1 USE: y, (2, 3)
DEF: y, 1 USE: y, (2, 6)
DEF: y, 1 USE: y, (3, 4)
DEF: y, 1 USE: y, (3, 5)
```

Both of these lists where generated by identifying all the definitions of each variable and then finding all the uses of the variable under consideration which can be reached by following the edges of the program graph.

Each line of the following two lists represents a definition use pair with respect to some variable `v` (*def-use pair wrt* `v`), which is an ordered pair containing a definition of variable `v` is defined and a use of `v`. (`v` is a free variable ranging over the variables declared in the program we are considering.) We will represent a def-use pair as a triple containing the definition, the use, and the variable. For example, the first two lines of the previous table of def-uses of `y` are represented as `< 1, 4, y >` and `< 1, (2, 3), y >`.

Notice that since every program has a finite number of nodes and edges, the number of def-use pairs is finite. We will make use of this fact later. When the context makes clear which variable `v` we are discussing, we will omit the "wrt `v`".

Having identified all possible def-use pairs for all variables, let's review the intuition underlying data flow testing and look at some examples. To apply data flow testing, we want to exercise paths which allow us to check that each variable is defined and used correctly. The application of data flow testing involves deciding which def-use pairs should be tested, constructing a set of paths which test those def-use pairs, and then constructing a test set which exercises that set of paths. This process will be discussed more carefully in the next section.

To conclude this section, let's construct a set of paths which tests all the def-use pairs of `y`. First, let's define a useful locution. *A path tests a def-use pair wrt* `v` when that path is a complete path which includes the definition of `v` and later, before `v` is defined by another statement in the path, includes the use. Notice the def-use pair `< 1, 5, x >` is tested by the path `( 1, 2, 3, 5 )`, but not by `( 1, 2, 3, 4, 3, 5 )` because `x` is redefined at node `4` before the use in node `5`.

Now let's commence construction.

The path `( 1, 2, 6 )` tests the def-use pair `< 1, (2, 6), y >`. The path `( 1, 2, 3, 5 )` tests the def-use pair `< 1, (2, 3), y >`. The path `( 1, 2, 3, 4, 3, 5 )` tests the three remaining def-use pairs: `< 1, 4, y >`, `< 1, (3, 4), y >`, and `< 1, (3, 5), y >`, as well as retesting `< 1, (2, 3), y >`.

As the `( 1, 2, 3, 4, 3, 5 )` path demonstrates, a single path can test many def-use pairs. All that is required is that the path include the definition and later, before the variable is redefined, all the uses.

Conversely, notice that for a given def-use pair, there may be many paths which include the definition and then before the variable is defined again, include that use. For example, the def-use pair `< 1, (3, 5), y >` is tested by both `( 1, 2, 3, 5 )` and `( 1, 2, 3, 4, 3, 5 )`.

So, the following set of paths tests all the definitions and uses of `y`:

```
{  ( 1, 2, 6 ),
   ( 1, 2, 3, 5 ),
   ( 1, 2, 3, 4, 3, 5 )
}
```

Also note the path `( 1, 2, 3, 5 )` is redundant, since `( 1, 2, 3, 4, 3, 5 )` tests the def-use pair tested by the first path. So, the following set of paths also tests all the definitions and uses of `y`.

```
{  ( 1, 2, 6 ),
   ( 1, 2, 3, 4, 3,  5 )
}
```

In general, many sets of paths may test the same def-use pairs, although these sets may have different elements or different numbers of elements.

In the next section, we make these ideas precise.

Data Flow Testing Criteria

Once we have identified all the def-use pairs in a program, the question arises: which should we test? There is more than one answer. Each answer defines a different data flow criterion. A data flow criterion specifies what kinds of paths need to be exercised for a program to be adequately tested with respect to that criterion. Rapps and Weyuker specify nine different data flow criteria and rate them by their thoroughness [RaWey85].

This section has two parts: First, we give algorithms for constructing sets of paths which satisfy three data flow criteria, and then, we give formal definitions of those data flow criteria in terms of graph theory.

We examine the three data flow criteria which are most commonly used: all-defs, all-p-uses/some-c-uses, and all-uses. We will introduce them in ascending order, with respect to their thoroughness. More specifically, for each criterion, we give an algorithm for constructing a set of paths which satisfies that criterion.

The following algorithms all assume that for every use of a variable, there is a path which contains a definition of that variable followed by that use. This assumption can be checked by a simple syntactic analysis. This assumption is meant to ensure that no variable is used before it is defined, but we will see later that the situation is not so simple.

Each of the algorithms have a common starting point which is described in the next paragraph. Start each algorithm with an empty set of paths `P`.

For each variable, consider each of its definitions. For each such definition, consider all the uses of that variable that are reachable by following the program graph before reaching another definition of that variable. From this set of uses, differentiate the c-uses and the p-uses.

From this point, the action we take is determined by which criteria we are trying to satisfy. To satisfy each criterion, we may need to select a different set of paths.

If we are applying the **all-defs** criterion, then for each definition of each variable:

- we must choose one use of that variable (either a c-use or a p-use) which can be reached from that definition without causing another definition of that variable.

- Next, find a path which tests the def-use pair composed of the definition and use we are considering. Add this path to the set of paths `P`.

Any set of paths `P` constructed in this way satisfies the all-defs data flow testing criterion.

Let's apply all-defs testing to the program in the Figure 2.1. For each of the three definitions, `DEF: x, 1`, `DEF: y, 1`, and `DEF: x, 4`, we must randomly choose a use which is reachable from that definition before the variable is defined again. Suppose we choose the following uses:

```
USE: x, (2, 3) to follow DEF: x, 1,
USE: y, (2, 3) to follow DEF: y, 1,
USE: x, (3, 5) to follow DEF: x, 4.
```

Then, the path `( 1, 2, 3, 4, 3, 5 )` tests all these def-use pairs. Therefore the singleton set containing that path satisfies the all-defs data flow criterion. Notice that we have been able to construct a set of paths which satisfies all-defs but does not exercise all the statements or branches in the program. We will address the question of rigorously comparing the thoroughness of the different data flow criteria later. Any other set which contains `( 1, 2, 3, 4, 3, 5 )` also satisfies all-defs.

Note we could have chosen `USE: y, (2, 6)` to follow `DEF: y, 1`, instead of `USE: y, (2, 3 )`. Had we done so, we would have been required to include the path `( 1, 2, 6 )`, which would have caused all the branches of the program to be traversed. The point is that the all-defs criterion allows us to choose any use of a variable which follows a particular definition of that variable, although some choices may lead to more thorough test sets.

If we are applying the **all-p-uses/some-c-uses criterion**, then for each definition of each variable:

- Determine if any p-uses of that variable are reachable from the definition we are considering, without that variable being redefined.

- If there are any such p-uses, then for each such p-use, find a path which tests the def-use pair composed of the definition and p-use under consideration. Add that path to the set of paths `P`.

- If there are no such p-uses, then identify all the c-uses which are reachable without defining the variable again. Randomly choose one of these c-uses. Next, find a path which tests the def-use pair composed of the definition and c-use under inspection. Add that path to the set of paths `P`.

Any set of paths `P` generated by following the above algorithm satisfies the all-p-uses/some-c-uses data flow testing criterion.

Again using the program from the Figure 2.1, we construct the following def-use pairs to test:

```
< 1, (2, 6), x >
< 1, (2, 3), x >
< 1, (2, 6), y >
< 1, (2, 3), y >
< 4, (3, 4), x >
< 4, (3, 5), x >
```

Notice that for each definition of each variable, there is a p-use which follows it, so there was no need to search for c-uses. A set of paths which tests all these def-use pairs and thus satisfies all-p-uses/some-c-uses is `{ ( 1, 2, 6 ), ( 1, 2, 3, 4, 3, 5 ) }`.

An example of a program for which there is definition which is not followed by a p-use is the following, which computes `x` div `y` for non-negative `x` and positive `y`:

```
quotient := 0;
readln( x, y );

if ( x >= 0 ) and ( y > 0 ) then begin
     while x >= y do begin
          x := x - y;
          quotient := quotient + 1
     end;

     writeln( quotient )
end else
     writeln( 'error' );
```

Figure 2.3

The reader should verify that the definition of `quotient` in the while loop is not followed by any p-use of that variable. Therefore, if we apply all-p-uses/some-c-uses to this program, the definition of `quotient` must be tested by a path which contains that definition and one of the two c-uses which follow that definition.

Finally note that Rapps and Weyuker's paper contains a program which has an error which is not caught by branch coverage but is caught by all-p-uses/some-c-uses.

If we are applying the **all-uses** criterion, then for all definitions of all variables:

- Identify all the uses of that variable which are reachable from the definition under consideration, without redefining the variable we are considering.

- For each such use, find a path which tests the def-use pair composed of the definition and use under consideration. Add that path to the set of paths `P`.

Any set of paths `P` generated by following the above algorithm satisfies the all-uses data flow testing criterion.

For the program in Figure 2.1, we construct the following def-use pairs to test:

```
< 1, 4, x >
< 1, 5, x >
< 1, (2, 6), x >
< 1, (2, 3), x >
< 1, (3, 4), x >
< 1, (3, 5), x >
< 1, (2, 6), y >
< 1, (2, 3), y >
< 1, (3, 4), y >
< 1, (3, 5), y >
< 4, 5, x >
< 4, (3, 4), x >
< 4, (3, 5), x >
```

The reader should verify that this set of paths tests all these def-use pairs and thus satisfies all-uses:

```
{  ( 1, 2, 6 ),
   ( 1, 2, 3, 5 ),
   ( 1, 2, 3, 4, 3, 4, 3  5 )
}
```

The all-p-uses/some-c-uses and all-uses criterion occupy the middle ground between statement and branch coverage on one hand, and all-paths testing on the other, which Rapps and Weyuker had set out to find. We will be able to appreciate why this is true when we discuss comparing the various criteria.

The reader can now appreciate why each criteria was given its name: All-uses checks all the uses which follow all the definitions of each variable. All-p-uses/some-c-uses checks all the p-uses, if any, which follow all the definitions of each variable, otherwise it checks some c-use for that definition. Finally, all-defs does little more than check one use after all definitions of all the variables.

In what remains of this section, we will give the graph theoretic definitions of the data flow criteria we have been studying, which are similar to the definitions proposed by Rapps and Weyuker [RaWey85]. To define the data flow criteria requires the following auxiliary definitions.

Suppose we have isolated a definition and a use of variable `v`. (Note that the use may either be a c-use or a p-use.) A definition clear path with respect to the def-use pair composed of `d` and `u` (*def clear path wrt* `< d, u, v >`) is a path, not necessarily complete, which begins with `d`, ends with `u`, and does not redefine `v` in any of the nodes between `d` and `u`.

Let `V` be the set of variables declared in the program. Let `N` be the set of nodes and `E` be the set of edges in the program graph. Define:

```
definition( v ) = { all nodes which define variable v }
```

(Note: Do not confuse this definition with Rapps and Weyuker's `def( i )`.)

```
c-use( n ) = { all variables which have c-uses in node n }
p-use( i, j ) = { all variables which have p-uses on
   edge ( i, j ) }

dcu( v, d ) = { all nodes u such that v is a member of
   c-use( d ) and there is a def-clear path with respect
   to v from d to u }
dpu( v, d ) = { all edges ( j, k ) such that v is a
   member of p-use ( j, k ) and there is a def-clear path
   with respect to v from d to ( j, k ) }
```

Let `C` be the set of complete paths and let `P` be a subset of `C`.

`P` satisfies **all-defs** if and only if:

```
for all variables v in V,
    for all nodes d in definition( v ),
        there exists a node u of dcu( v, d ) such that
            P has a member which contains a sub-path which is
            a def clear path wrt < d, u, v >
        or
        there exists an edge u of dpu( v, d ) such that
            P has a member which contains a sub-path which is
            a def clear path wrt < d, u, v >
```

`P` satisfies **all-p-uses/some-c-uses** if and only if:

```
for all variables v in V,
    for all nodes d in definition( v ),
        dpu( v, d ) is empty implies
            there exists a node u of dcu( v, d ) such that
                P has a member which contains a sub-path which is
                a def clear path wrt < d, u, v >
        and
        dpu( v, d ) is not empty implies
            for all edges u in dpu( v, d )
                P has a member which contains a sub-path which is
                a def clear path wrt < d, u, v >
```

`P` satisfies **all-uses** if and only if:

```
for all variables v in V,
        for all nodes d in definition( v )
                for all edges u in dpu( v, d )
                        P has a member which contains a sub-path which is
                        a def clear path wrt < d, u, v >
                and
                for all nodes u in dcu( v, d )
                        P has a member which contains a sub-path which is
                        a def clear path wrt < d, u, v >
```

Infeasible Paths and Data Flow Testing

We now turn to a significant problem shared by all path testing criteria: Suppose a tester has selected a data flow testing criterion. For many programs, while the tester can construct a set of paths which satisfies that criterion, there are no test sets which cause that set of paths to execute. How can this be? The answer is that the tester chose a path which no input datum can exercise. We explore this problem in this section.

Before we continue, let's reexamine our definition of path. Recall our discussion of path testing began with an intuitive notion of path. We then formalized this notion in the language of graph theory.

However, there is a discrepancy between the intuitive and the graph theoretic notions of a path. Recall that the intuitive idea of a path is that it is the list of the simple statements which are executed when the program is run with some input datum. In contrast, the formal notion of a path is a list of nodes which is consistent with the program graph. The discrepancy is this: there are paths, in the graph theoretic sense, which do not correspond to any paths, in the intuitive sense. More specifically, for some programs, there exist lists of nodes which are consistent with the program graph, but for which there exists no input data which causes that list of simple statements to be executed. We saw an example of this in our discussion of the all-paths testing criterion. We find another example in the following program fragment and its program graph.

```
readln( x, y );

if x = 0 then
     y := 1;
else
     y := 0;

if ( x = 0 ) and ( y = 0 ) then
     writeln( 'never' )
else
     writeln( 'always' );
```

Figure 2.4

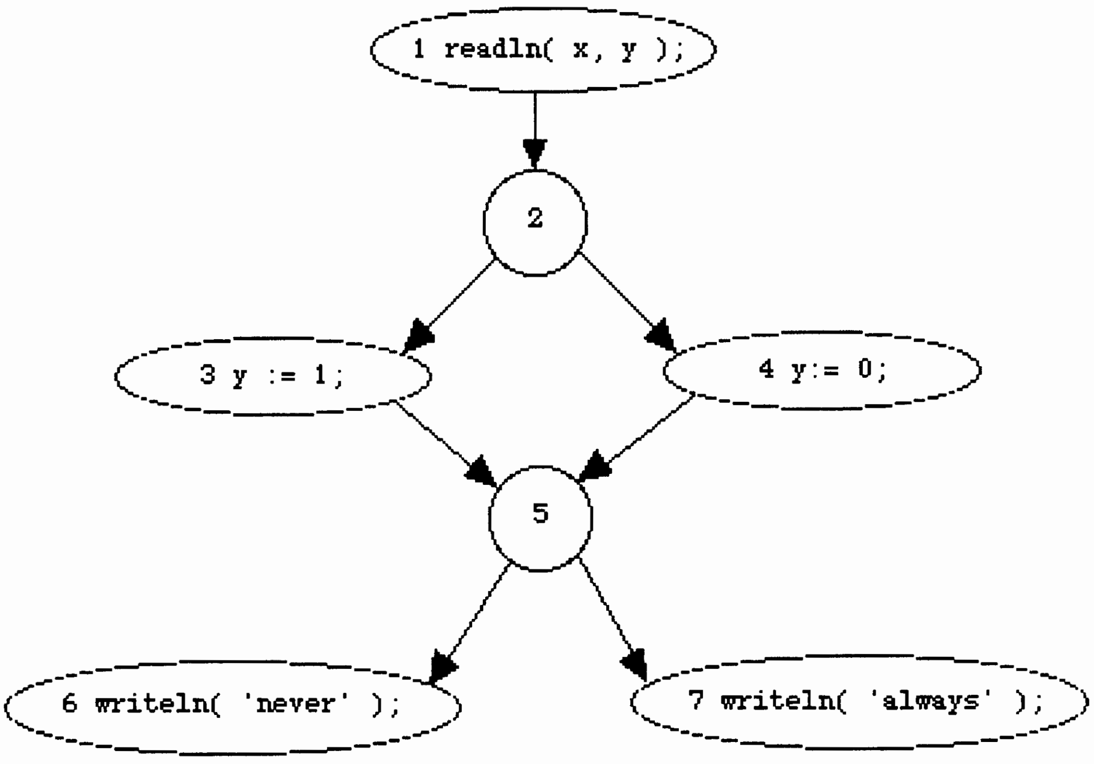

Figure 2.5

Notice that there is no input datum which can make the boolean expression in the second **if** statement evaluate to true and display 'never'. If we have our intuitive idea of path in mind, we say that there is no path which contains the statement which displays 'never'. However, in the graph theoretic sense, `( 1, 2, 3, 5, 6 )`, for example, is a perfectly acceptable path. A path which is not exercised by any input datum is called "infeasible", while a path for which there exists input data which cause that path to be exercised is called "feasible". Note that it is a contradiction in terms to speak of the execution of an infeasible path.

Thus, if we attempt to apply all-p-uses/some-c-uses to the above program, we will be required to exercise a complete path which includes a def-clear path with respect to the def-use pair `< 3, (5, 6), y >`. However, by the previous argument, there is no input datum which can exercise such a path. Therefore, there is no way to apply the all-p-uses/some-c-uses data flow testing criterion, as we defined it in the previous section, to this program.

Note that although the infeasible path in the above example is unlikely to appear in real programs, Frankl and Weyuker assure us there are many "reasonable" programs contain infeasible paths [FraWey88]. Thus, due to the existence of infeasible paths, there are programs which we cannot test with a given data flow criteria.

The next section deals with Frankl and Weyuker's attempt to circumvent the problem of infeasible paths.

Feasible Data Flow Criteria

The problem presented in the previous section is that for any data flow criterion, we may construct a set of paths which satisfies that data flow criterion, but for which there is no test set which exercises those paths. The source of this problem is that the data flow criteria do not differentiate between feasible and infeasible paths.

Frankl and Weyuker define a new family of data flow criteria which are just like the data flow criteria proposed by Rapps and Weyuker, except that these new criteria require that every def-use pair must be tested by a feasible path [FraWey88]. If a def-use pair cannot be tested by a feasible path, then it can be ignored. These new data flow criteria are called *feasible data flow criteria*.

More specifically, for each original data flow criterion, there is a corresponding feasible data flow criterion. Further, each feasible criterion has the same name as its associated original criterion, followed by an asterisk. So, all-defs*, for example, is the feasible data flow criterion which corresponds to the original all-defs data flow criterion.

We now give informal definitions for all-defs*, all-p-uses/some-c-uses*, and all-uses*. In general, these definitions will be the same as the definitions for the original data flow criteria, except that we will require that each path which tests a def-use pair be feasible. Note we defer issues regarding unsolvability until the next section. We begin by following the same steps we followed for defining the original data flow criteria.

For each variable, consider each of its definitions. For each such definition, consider all the uses can be reached by following the program graph before reaching another definition of that variable. From this set of uses, differentiate the c-uses and the p-uses.

From this point, the action we take is determined by which criteria we are trying to satisfy.

If we are applying the **all-defs*** criterion, then for each definition of each variable:

- We must randomly choose one use of that variable (either a c-use or a p-use) which has two properties: 1) it can be reached from that definition without causing another definition of that variable and 2) there exists a feasible path which tests the def-use pair composed of the definition and use we are considering.

- If such a use exists, then add one of these feasible paths to the set of paths.

- If there is no use exists (or no such feasible path exists), then we add nothing to the set of paths.

Any set of paths constructed in this way satisfies the all-defs* data flow testing criterion.

If we are applying the **all-p-uses/some-c-uses criterion***, then for each definition of each variable:

- Determine if any p-use of that variable has the two properties that : 1) it is reachable from the definition we are considering, without that variable being redefined and 2) there exists a feasible path which tests the def-use pair composed of the definition and p-use under consideration.

- If such a p-use exists, then add one of these feasible paths to the set of paths.

- If there are no such p-uses (or no such feasible paths), then identify all the c-uses which are reachable without defining the variable again and for which there exists a feasible path which tests the def-use pair composed of the definition and c-use under inspection. Randomly choose one of these feasible paths and add it to the set of paths.

Any set of paths generated by following the above algorithm satisfies the all-p-uses/some-c-uses* data flow testing criterion.

If we are applying the **all-uses*** criterion, then for all definitions of all variables:

- Identify all the uses of that variable which: 1) are reachable from the definition under consideration, and 2) for which there exists a feasible path which tests the def-use pair composed of the definition and use under consideration.
- Add one of these feasible paths to the set of paths.

Any set of paths generated by following the above algorithm satisfies the all-uses* data flow testing criterion.

Frankl and Weyuker show that for every feasible data flow criteria and every program, there exists a test set which satisfies that criterion for that program. In this sense, they solved the problem created by infeasible paths. However, the feasible data flow criteria have their own set of unpleasant problems. We will consider these problems soon.

The formal graph theoretic definitions of these criteria require two new definitions:

```
fdcu( v, d ) = { all nodes u such that v is a member of
  c-use( d ) and there is a feasible def-clear path with
  respect to v from d to u }

fdpu( v, d ) = { all edges ( j, k ) such that v is a
  member of p-use ( j, k ) and there is a feasible def-
  clear path with respect to v from d to ( j, k ) }
```

To obtain the definitions of the feasible data flow criteria from the definitions of the original data flow criteria, simply replace each instance of functions `dcu` and `dpu` in the original definitions with the functions `fdcu` and `fdpu`, respectively and replace each reference to a def clear path with a reference to a feasible, def clear path.

The feasible data flow criteria have two problems, a small one and a large one. The small problem is that any programming error which creates an infeasible path will be ignored by all our feasible criteria. The big problem is discussed in the next section.

Unsolvability and Feasible Data Flow Criteria

We have seen that the feasible data flow criteria were proposed to remedy a problem concerning infeasible paths in the original data flow criteria. Basically, the feasible data flow criteria are identical to the original data flow criteria except that they ignore infeasible paths. Unfortunately, the feasible data flow criteria have a problem at least as serious as the problem they were invented to solve: there exist paths which we cannot identify as feasible or infeasible. Note that, granted the law of the excluded middle, each path truly is either feasible or infeasible, but there exist paths for which we cannot determine the actual case. This is the issue we explore in this section.

We begin by discussing the general notion of unsolvability (also called undecidability), then we observe that identifying an arbitrary path as feasible or infeasible is an unsolvable problem. Finally, we investigate the implications of this observation for feasible data flow testing.

A problem is unsolvable if we can demonstrate, using the methods of mathematical logic, that an algorithmic solution to that problem does not exist [ManGhe87]. Such a demonstration usually takes the form of a reductio ad absurdum: For a given (unsolvable) problem, we assume there exists an algorithmic solution to the problem and then from that assumption we derive a contradiction. Because of our faith that contradictions do not exist in logic and mathematics, we conclude that the supposed algorithmic solution does not exist. Thus, the problem which the algorithm was supposed to solve is labelled unsolvable.

The fact that a problem is unsolvable in general does not mean that we can't solve particular instances of that problem. It just means that there can be no algorithm which solves the problem for all cases. We consider some examples of this.

The unsolvable problem which Computer Scientists are most familiar with is the halting problem, which establishes that there is no algorithm which takes any program and any input datum as input and (correctly) decides whether or not that program terminates when applied to that input datum. However, note that the halting problem can be solved for many particular programs. A trivial example of this is the set of programs which contain no loops or function calls. All the programs in this set can be proven to terminate for all inputs. Nonetheless, the halting problem is unsolvable for the class of all programs.

Unsolvability will be our constant companion throughout this thesis. However, all the instances of unsolvability which we will encounter can be understood as special cases of the following result: Given an expression E, composed of boolean and arithmetic operations and constants, and which contains at least one free variable v which ranges over an infinite domain, there is no algorithm which can decide whether or not there is an assignment to v such that E is true. A proof of this result can be found in Alonzo Church's classic paper *A Note on the Entscheidungsproblem* [Church36].

This result does not contradict the fact that for many expressions we can either construct the required assignment or a proof that no assignment exists. Instead, it demonstrates that we cannot construct an algorithm which will work for all expressions. If we think we have constructed such an algorithm, then either it does not work correctly in all cases, or for some inputs it never terminates.

In practical terms, the aforementioned unsolvability result tells us the following: Suppose we have an expression for which we are trying to find an assignment which satisfies it or a proof that there is no such assignment. Although (we assume that) there either exists such an assignment or there does not, we have no guarantee that, if we work on the problem for some finite amount of time, we discover which is the case. Maybe we will find the answer in our next attempt. Maybe we will try for years and fail to find it. (Maybe if we would have tried just another minute, we would have found the answer.) To those who find this situation contrived or unlikely to occur in "real life", we encourage you to continue reading.

We now return our attention to infeasible paths and observe that detecting infeasible paths is an unsolvable problem [Hamlet88]. Here's why: Suppose we have a path `P` and we want to decide if it is infeasible or not. Recall that the input data which causes a path to be exercised can be characterized by a logical expression. Let `E` be the expression which characterizes the input data which exercise `P`. The question of whether `P` is infeasible or not is just the question of whether there is an assignment to the free variables of `E` of some input datum which causes `E` to evaluate to true or there is a proof that no such input datum exists. We have just observed that this problem is unsolvable. Hence the detection of infeasible paths is an unsolvable problem.

The fact that detecting infeasible paths is an unsolvable problem cripples all the data flow testing criteria we have so far considered. In what remains of this section, we will consider how unsolvability affects the original data flow criteria and the feasible data flow criteria.

For any given original data flow criterion, while constructing a set of paths which satisfy that criterion presents no problem, finding a set of input data which exercises those paths is in general unsolvable. For any feasible data flow criterion, constructing a set of paths which satisfies the criterion is unsolvable, however, if such a set can be identified, then constructing a set of paths which exercises those paths is not difficult.

In conclusion, we have no choice but to abandon the original data flow criteria, because these criteria cannot test programs with infeasible paths. However, the feasible data flow criteria require the tester to face the unsolvable problem of identifying feasible and infeasible paths. In the chapters which follow, we shall only be concerned with the feasible data flow criteria.

3 Program Verification Formalisms

This thesis involves program verification formalisms in two ways: First, data flow testing is usually supported by a program verification formalism called *symbolic execution* [HanKin76]. We will see that symbolic execution cannot easily be applied to array elements due to unsolvability. This limitation consequently limits the data flow testing systems which depend on symbolic execution. Second, this thesis presents a method for alleviating the aforementioned problem based on the program verification formalism called *weakest pre-condition*, invented by Edsger W. Dijkstra [Dijkstra76]. This chapter devotes a section to each formalism.

Symbolic Execution and Data Flow Testing

In this section, we review symbolic execution, discuss its application to data flow testing, and examine why it has difficulty with array variables.

Symbolic Execution

The intuition behind symbolic execution is expressed well by its creators [HanKin76]:

> One can use a standard mathematical technique of inventing symbols to represent arbitrary program inputs, and then attempt a proof involving those symbols. If no special properties of the symbols, other than those expected to hold for all inputs, are necessary for the proof, then the proof is valid for each specific input. If special properties of the symbols must be assumed in order to construct a proof, then an exhaustive case analysis can be performed, providing a set of proofs, one for each case, which collectively give a complete proof.

In this section, we expand on the above summary enough to motivate the problem of this thesis.

To understand the basic methodology of symbolic execution, we need to discuss: program states, program specification, symbolic values, and symbolic execution states. Our knowledge of paths will also be useful.

A program state is a record of the current contents of each variable during the actual (as opposed to symbolic) execution of a program. In the section on Dijkstra's weakest pre-condition formalism, we offer a more formal definition, but this intuitive notion suffices for now.

The first step in the employment of symbolic execution to prove the correctness of a program is to formally specify that program. In the case of symbolic execution, formal specification of a program entails the construction of two logical expressions: one which describes the assumptions, if any, which we make about the input state and the other which describes the conditions which the variables should satisfy if the program terminates. The former is called a *pre-condition* or an input assertion and the latter is called a *post-condition* or an output assertion. Symbolic execution allows us to demonstrate the correctness of a program by proving that, for any initial state which satisfies the pre-condition, if the program terminates when applied to that initial state, the program will terminate in a state which satisfies the post-condition. Note that symbolic execution does not prove that the program will terminate, thus we need to always say, "if the program terminates".

As Hantler and King mentioned in their summary above, symbolic execution uses symbols to represent the contents of the variables in the program. This is the distinctive feature of symbolic execution. We will use the names of Greek letters in an underlined `Courier` font for this purpose, for example, `alpha` and `beta`. The symbols used to represent the contents of the variables when the program begins execution will be called the *symbolic values of the program.* Note that a symbolic execution of a program can be transformed into an actual execution by replacing each symbolic value with an actual value of the correct type, just as an algebraic computation can be transformed into an arithmetic computation by replacing the algebraic variables with actual numbers.

A *symbolic expression* is an expression which includes at least one symbolic value, for example, `alpha` , `alpha + 1`, or `alpha * beta`.

When we apply symbolic execution to a program, each variable in that program is allowed to contain a symbolic expression as its contents. Thus, if one of the variables in our program is `x`, then `x` can contain the symbolic expression `2 * alpha + beta - 1`, for example.

We can now discuss how assignment statements are evaluated in symbolic execution. When symbolic execution is applied to an assignment statement, the expression on the right hand side of the assignment operator is evaluated, and the resulting value, possibly symbolic, is copied into the variable on the right hand side of the assignment operator. Now we need to explain how an expression is evaluated under symbolic execution.

Evaluating an expression, in the context of symbolic execution, involves two steps:

1. Replace each variable with its contents, possibly symbolic.
2. Perform algebraic simplifications as desired.

For example, suppose `x` has as its contents the symbolic expression `2 * alpha + beta + 1` and `y` has for its contents `alpha - 1`. Then the assignment statement `z := x - y;` is evaluated in the following manner: First, the variables in `x - y` are replaced by their contents, yielding `( 2 * alpha + beta + 1 ) - ( alpha - 1 )`. Then we simplify the expression to `alpha + beta + 2`. We then copy this symbolic expression into the contents of the variable `z`.

When we execute a real program on a real machine, the current state of the program indicates the value that each variable contains. There is an analogous notion in symbolic execution. A *symbolic execution state* has two components:

1. the current contents of each variable, which in the case of symbolic execution, can be a symbolic expression,

2. the *path condition*, which is a logical expression which constrains the values which the symbolic values of the program can take. The current path condition characterizes all the input values which cause the path under consideration to be executed.

We now describe how symbolic execution works for programs without loops. This limited discussion will be adequate for our purposes. At the end of this section, we will briefly consider how symbolic execution handles loops.

Symbolic execution begins by constructing a program graph, just like what we discussed in Chapter one, except in this context, it is called an *execution tree.* Then, we apply the following algorithm to each complete path through the execution tree. If the application of this algorithm is successful for each path, then we conclude that, for any initial state which satisfies the pre-condition, if the program terminates when applied to that initial state, the program will be in a state which satisfies the post-condition. If the following algorithms fails for even one path, then the program is not verified with respect to its formal specification.

Repeat the following procedure until every complete path has been traversed:

- Initialize the variables with unique symbolic values and initialize the path condition to the pre-condition.

- If an assignment statement is encountered, replace the contents of the variable on the left hand side of the assignment operator with the value, possible symbolic, to which the expression on the right hand side of the assignment operator evaluates.

- If an **if** statement is encountered, then choose a branch to follow which hasn't been traversed yet. Modify the path condition to reflect the constraints on the symbolic values which must hold for this branch to be followed. This modification involves replacing the variables in the boolean expression with their values, possibly symbolic, and conjoining the resultant logical expression with the previous path condition.

  Just as assignment statements can change the contents of variables, the boolean tests that control **if** and **while** statements can change the path condition.

- If the end of the path is encountered, then a theorem must be proved which establishes that the traversal of this path satisfies the post-condition. To construct that theorem, first replace all the variables in the post-condition with the contents of those variables at the end of this symbolic execution. The theorem to be proven states that the post condition logically implies the expression constructed in the previous sentence. If this proof cannot be carried out for some path, then the verification fails.

Note that symbolic execution proceeds by traversing each path from beginning to end and, at the end of each path, proving a theorem. We will later contrast this approach with Dijkstra's symbolic execution.

Let's examine a simple example which puts the absolute difference of the variables `x` and `y` into the variable `abs_diff`.

```
if x >= y then
     abs_diff := x - y
else
     abs_diff := y - x;
```

Figure 3.1

The first thing we must do is formally specify the program. Since we don't need to put any constraints on any of the values of the variables, our pre-condition is simply the predicate `true` which returns the truth value `true` for every state (Note that we have followed the convention of overloading the symbol "`true`".) Our post-condition will be `abs_diff >= 0 and ( abs_diff = x - y or abs_diff = y - x )`. Note that a complete post-condition would also specify that the code not change the values of `x` and `y`. For our simple purposes, we can ignore this requirement.

The next step of the symbolic execution is to construct the execution tree. We will not display this tree, because it is obvious.

Now, we set the path condition to the pre-condition, which is the predicate `true`, and set the variables `x` and `y` to the symbolic values alpha and beta, respectively.

There are two paths to traverse: the path where the boolean expression which controls the **if** statement evaluates to `true` and the path where the expression evaluates to `false`.

To traverse the path where the boolean expression evaluates to `true`, we modify the path condition to reflect what must be true of the symbolic values for the expression to evaluate to `true`. So the path condition becomes `alpha >= beta`. Then we encounter the assignment statement `abs_diff := x - y`. This statement causes the contents of `abs_diff` to be changed to `alpha - beta`. Now we are at the end of the first path, so we must prove that traversing that path satisfies the post-condition. The theorem we need to prove is constructed by first replacing all the variables in the post-condition with their contents at the end of the path. Carrying out this replacement, we receive `alpha - beta >= 0 and ( alpha - beta = alpha - beta or alpha`

- beta = beta - alpha). We can simplify this to just alpha - beta >= 0.The theorem we need to prove is that the path condition at the end of the symbolic execution implies the expression we just constructed. Thus, the theorem we need to prove is:

```
    alpha >= beta => alpha - beta >= 0
```

which is trivial. Thus, the verification has succeeded for this path. We would verify the other path in an analogous manner and conclude that the program is verified with respect to its formal specification.

We have covered the material necessary to appreciate the problem which symbolic execution has with arrays. However, recall that, in the interest of brevity and simplicity, we did not consider how to apply symbolic execution to programs with loops. We conclude this section with a short discussion of this neglected topic.

Symbolic execution traverses all the paths of the execution tree and proves that at the end of each path the post-condition is satisfied. However, as we discovered in our discussion of all-paths testing in Chapter two, an execution tree which contains a loop has an infinite number of paths. This is a problem, because a person or machine employing symbolic execution to verify a program cannot traverse an infinite number of paths. The solution to this problem is an inductive technique for traversing the paths which, for each loop in a program, produces a logical expression which will be true regardless of how many times the loop iterates. This induction obviates traversing an infinite number of paths. This technique is conceptually identical to Hoare's loop invariants [Hoare69], and like Hoare's invariants, has yet to be mechanized.

Having reviewed symbolic execution, we now turn to the relationship between symbolic execution and data flow testing.

Symbolic Execution in Support of Data Flow Testing

Symbolic execution allows us to produce a logical expression which characterizes the input states which cause a specified path to be exercised: Just follow the algorithm explained in the last section until you reach the end of the path. The path condition will contain a predicate which characterizes all the input states which exercise that path.

Symbolic Execution and the Problem of Array Variables

Having just discussed symbolic execution and data flow testing, we turn to the problem which motivates this thesis: only with difficulty can symbolic execution be applied to reason about array elements.

Once again, the problem is unsolvability. In particular, consider the assignment statement `a[ i ] = 0;` in the middle of a complicated program. Determining which array element is defined in that statement is an unsolvable problem.

While unsolvability did not deter Frankl and Weyuker from proposing the feasible data flow criteria, it did deter many developers of symbolic execution from considering programs with arrays. We will see that the wp_du method "makes the best" of this unsolvable problem by reducing this unsolvability to a problem for which there are many partial solutions.

Hamlet et. al. [HamGN93] review the methods by which most developers of symbolic execution have attempted to handle arrays and summarize the reasons that these attempts are unsatisfactory.

To avoid the problems presented by unsolvability, researchers in data flow testing decided to ignore the fact that an array has individual elements and instead, treat any definition of any array element as a definition of the entire array and any use of any array element as a use of the entire array. As Hamlet et. al. observe:

> Data-flow testing systems have treated program arrays as aggregate objects that are "set" by assignment to any element and "used" when reference is made to any element. This treatment is clearly worst-case, and not a very good approximation to what data-flow tests intuitively cover.

It is not surprising that this simplified method is imprecise. In fact, it makes errors of commission and omission: That is, it requires that we test def-use pairs where the definition defines a different array element than the use uses and it ignores def-use pairs that should be tested. Consider the following code:

```
a[ 1 ] := 1;

if x > 0 then
     writeln( a[ 2 ]  );
```

If we treat the array as an aggregate object, then we are forced to say that the same variable is defined in the assignment statement and used in the output statement. But clearly, the array element that is defined is not the array element that is used. Any data flow criterion would require that we test for a data dependency between the definition and use, but clearly, this is wasted effort.

In addition to requiring wasteful tests, the method of treating arrays as aggregate objects also fails to require that essential tests be conducted. For example, consider the code fragment which exchanges two distinct elements in array a, using a[1] as temporary:

```
a[ 1 ] = a[ i ];
a[ i ] = a[ j ];
a[ j ] = a[ 1 ];
```

Further suppose that `i` and `j` will not be `1`. Then we can see that the only array element which is defined and then used in this code is `a[ 1 ]`. However, if we interpret the array as an aggregate object, then we are forced to say that the array variable is defined in the first statement and used in the second, and defined in the second statement and used in the third. Therefore, the method we are considering finds two data dependencies which do not exist and overlooks the one that does exist.

In summary, if we want to test a program with arrays using a data flow criteria, the only available method is to treat each array as a single variable, which corrupts the data flow analysis.

This thesis addresses this problem, not by attempting another extension of symbolic execution to include arrays, but by applying the weakest pre-condition formalism.

Weakest Pre-Condition

In this section, we discuss Dijkstra's weakest pre-condition formalism [Dijkstra76]. This formalism has two complementary uses: program verification and program synthesis. It also is an elegant tool for investigating non-deterministic programs. Only the program verification faculty is relevant to our discussion.

The steps in proving that a program is correct using the weakest pre-condition formalism are as follows: First formally specify the pre-condition and the post-condition. Then apply the weakest pre-condition formalism, which is actually a set of higher-order functions, each of which takes a post-condition as an input. The weakest pre-condition formalism mechanically computes a logical expression which characterizes all the input states which will cause the program to terminate in a state satisfying the post-condition. For reasons which will be explained shortly, this logical expression is called the "weakest precondition with respect to that program and that post-condition". The program is verified if the pre-condition which was specified originally logically implies the weakest precondition.

In this section, we first explain the weakest pre-condition formalism, then explore different methods for representing the weakest precondition.
One method we discuss is power functions [Antoy87].

To begin with, we need to define some terminology.

In the earlier part of this chapter, we gave an intuitive definition of the state of a program. For our current discussion, we require a more formal definition. A *state* is a function whose domain is the the set of all possible variable names and whose range is the disjoint union of the set of values of each type in the language. A state must map a variable to a value of the same type.

A *predicate* is a function which takes as input a state and returns `true` or `false`. We will use the terms "logical expression" or "boolean expression" as synonyms for the term "predicate".

A *pre-condition* and a *post-condition* are both predicates over states.

A *predicate transformer* is a function whose domain and range are the set of all possible predicates. These functions are called "predicate transformers" because they take a predicate as input and transform it into the predicate which is the output of that function. The predicate transformers which we are interested in are those which take post-conditions as inputs and compute weakest pre-conditions as outputs. More specifically, each possible statement in our programming language will have a predicate transformer associated with it such that for a given post-condition, each predicate transformer will return the weakest pre-condition with respect to the specified statement and post-condition.

Recall that the weakest pre-condition with respect to a statement and a post-condition is the predicate which characterizes all the input states which cause the statement to terminate in a state which satisfies the post-condition. A pre-condition is weakest with respect to a statement and a post-condition when any further constraint added to that pre-condition would cause the predicate to fail to characterize all the input states which cause the statement to terminate in a state which satisfies the post-condition.

Dijkstra's weakest pre-condition formalism is a set of schemata for predicate transformers which compute weakest pre-conditions. In this context, a schema is a rule for generating predicate transformers which compute weakest pre-conditions. (In most other contexts, schema are used to generate axioms.) More specifically, a schema is a definition of a predicate transformer which contains free variables, such that when these free variables are instantiated, a predicate transformer which computes weakest pre-conditions is created.

We now present the weakest precondition schemata for our subset of Pascal. Note that Dijkstra defined his weakest precondition formalism for a language which had more general **if** and **while** statements than we use in this thesis.

Let `P` a free variable ranging over all post-conditions.

The schema for the **assignment statement predicate transformers** is:

Let `v` be a variable and `e` be an expression of the same type as `v`.

```
wp( v := e, P ) =
( P with e textually substituted for v )
```

When `v` and `e` are instantiated, a predicate transformer for that assignment statement is produced.

The schema for the **if statement predicate transformers** is:

Let `t` be a boolean test, and let `s1`, and `s2` be statements in the programming language.

```
wp( if t then s1 else s2, P ) =
( t => wp( s1, P )) and ( not t => wp( s2, P ))
```

When `t`, `s1`, and `s2` are instantiated, a predicate transformer for that **if** statement is generated.

The schema for the **while statement predicate transformers** requires some discussion. For every possible loop, the weakest precondition formalism associates an infinite sequence of predicates, where the n^{th} element of the sequence is the weakest pre-condition which causes the loop to iterate exactly `n` times. The weakest precondition for the loop states that there exists a natural number `k` such that the loop iterates `k` times and that the k^{th} element of the sequence conjoined with the loop test is the weakest pre-condition.

This sequence of weakest preconditions is represented by $\{\ H_k(P)\ \}$, for all natural numbers `k`.

Let `t` be a boolean test, and let `s` be a statement in the programming language.

```
wp( while t do s, P ) = there exists a natural
number k such that $H_k$( P ) is true where

$H_0$( P ) = not t and P
$H_{k+1}$( P ) =
t and wp( s, $H_k$( P )) for any natural number k
```

Note that Dijkstra does not give a soundness proof to establish that his schemata of predicate transformers do in fact always compute the weakest pre-condition. Presumably, he omitted such a proof because he intended the predicate transformers to be the most basic semantic description of the programming language.

Dijkstra's definitions comprise an effective procedure (an algorithm guaranteed to terminate on all inputs) for constructing a representation of the weakest pre-condition of a program with respect to a post-condition. This is the case because a program can be interpreted as a composition of statements, and such a composition of statements yields a composition of weakest pre-condition predicate transformers.

Further, note that the weakest precondition formalism begins with a post-condition and mechanically "works backward to the beginning of the program" to produce the weakest pre-condition. This is in contrast to symbolic execution, which starts at the beginning of the program and traverses all the complete paths to the post-condition. This difference will be important in the last section of chapter six.

Let's examine a short example, which will demonstrate what we have just discussed as well as motivate the need for alternative representations of the weakest pre-condition.

```
sum := 0;
i := 1;
while i <= n do begin
     sum := sum + i;
     i := i + 1
end;
```

Figure 3.2

We specify the pre-condition to be `0 <= n` and the post-condition to be `sum = n*(n + 1)/2`.

The weakest precondition can be represented as

```
(1)  wp( sum := 0; i := 1,
     there exists k such that Hk( sum = n*(n + 1)/2 ))
```

Note that the free variables `t`, `s1`, and `s2` in the definition of the { H_k(P) } sequence must be instantiated with the appropriate code from the program.

Although (1) is a correct representation of the weakest precondition, it is not a useful one. The problem is that neither humans nor mechanical solvers can easily manipulate the predicate transformers. Said plainly, (1) represents the set of states which cause the program to terminate in a state satisfying `sum = n*(n + 1)/2`, but we cannot easily identify any of the members of that set.

This problem motivates us to search for other means of representing the weakest pre-condition. In what remains of this section, we discuss two such alternative methods: the ingenuous method and the power function method. These alternative methods are used in Chapter six.

The **ingenuous method** involves finding an infinite sequence of predicates { G_k } such that for every natural number n, G_n = H_n(P) and no element of { G_k } involves a predicate transformer. The name of the method, "ingenuous", is taken from Loeckx and Sieber's work [LoeSie87]. The name is appropriate because there really is no method, other than unmechanized intuition, for generating the { G_k } sequence. Once we have invented such a sequence { G_k }, we prove that it is equivalent to the sequence { H_k(P) }, and then we use { G_k } instead of { H_k(P) }.

Turning to the code in Figure 3.2, the following infinite sequence of predicates { G_k } is equivalent to the sequence of predicates { H_k(sum = n*(n + 1)/2) }:

```
G₀ = H₀( sum = n*(n + 1)/2 )
```

for all natural numbers k

```
Gk+1 =
(i + k = n) and (sum + (k + 1)*i +
k*(k + 1)/2 = n*(n + 1)/2 )
```

(Note that the '=' symbol is overloaded: it represents both equality between natural numbers and definition of predicates.)

Note that a proof that { G_k } = { H_k(sum = n*(n + 1)/2) } would be inductive, where we first establish that G_1 = H_1(sum = n*(n + 1)/2), and then show that if, for some natural number k, G_k = H_k(sum = n*(n + 1)/2), then G_{k+1} = H_{k+1}(sum = n*(n + 1)/2). We will see an example of this kind of proof in Chapter six.

Once we have established that { G_k } = { H_k(sum = n*(n + 1)/2) }, then we can replace any element of the { H_k(sum = n*(n + 1)/2) } sequence in (1) with the associated element of the { G_k } sequence. Since none of the elements of the { G_k } sequence contain predicate transformers, the representation of the weakest pre-condition of the program in figure 3.2 is considerably simplified.

(1) becomes:

```
wp( sum := 0; i := 1, there exists k such that Gk )
```

If k is 0, then the wp predicate transformer yields n = 0 or n = -1.

If k is positive, wp yields

```
there exists a k such that k = n and
k * ( k + 1 ) / 2 = n * ( n + 1 ) / 2
```

which reduces to 0 < n.

Since k is either zero or positive, the weakest precondition for the program is -1 <= n. The program is verified if our pre-condition logically implies the weakest pre-condition. Since our weakest pre-condition, 0 <= n logically implies -1 <= n , the program is verified.

Recall that the problem with the ingenuous method is that it is not mechanizable. Since weakest pre-condition computations quickly overwhelm the ability of the human brain to manage complexity, researchers searched for another representation of the weakest pre-condition.

A problem shared by both Dijkstra's definition of weakest pre-condition and the ingenuous method is that each have features which prohibit mechanization. Antoy's research [Antoy87] has produced a representation of the weakest pre-condition which is amenable to mechanization. He focuses on the weakest preconditions for **while** statements, since they are the most difficult. His method is called the **power function method**.

In essence, his method expresses the weakest pre-condition of a loop in terms of first-order logic, in contrast to Dijkstra's predicate transformers, which belong to higher order logic. Expressions in first order logic are easier for machines to manipulate.

Before we can state his representation of the weakest pre-condition of a loop, we need to discuss: representing a state as a tuple, functional abstraction, and power functions.

In this thesis, we have already represented program states as sets of bindings and as functions. In this context, it is most convenient to conceive of a state as being a tuple, where each field in the tuple corresponds to a particular variable such that the field contains the value of the variable. Referring back to the program in Figure 3.2, we can represent the state as a triple, where, for example, the first field contains the value of `i`, the second field contains the value of `sum`, and the third field contains the value of `n`.

The functional abstraction of a code fragment is the function which accepts a state as input (represented by a tuple) and alters that state in exactly the same way that the code fragment alters the state of the machine. This is the idea upon which denotational semantics is founded. Suppose that the symbol `code` denotes a code fragment. Then `[code]` denotes the functional abstraction of that code. For example, the functional abstraction of the loop body of the program in Figure 2.3 can be represented as:

```
(2)   ( i, sum, n ) -> ( i + 1, sum + i, n )
```

where -> means "is mapped to".

Applying the functional abstraction to a state is thus analogous to executing the code with that state loaded into the machine's memory. Power functions help us express the idea of executing some code repeatedly, which allows us to model the execution of a loop that contains that code in it's body. More specifically, given a functional abstraction `[code]`, we represent the power function associated with `[code]` as `[code]*`. A power function takes two inputs: a natural number expressing the number of times we apply the functional abstraction and an initial state.

Formally, let `k` be a natural number and `s` be a tuple representing the state.

(3)

$$[code]^*(k, s) = \begin{cases} [code]^*(k-1, s) \text{ if } k > 0 \\ s \text{ if } k = 0 \end{cases}$$

We can now explain how power functions are used to represent the weakest pre-condition. Intuitively, the weakest pre-condition of a loop with respect to a post-condition `P` is a predicate which characterizes all the states which cause the loop to terminate in any state satisfying the post-condition. Expressed more carefully, this weakest pre-condition characterizes all the states `s` such that a natural number `k` exists which represents the number of times the loop iterates and that when the functional abstraction of the loop body is applied to state `s` `k` number of times, the resultant state will satisfy the post-condition.

Expressed symbolically, the weakest pre-condition of a loop of the form

```
while t do b
```

with respect to the post-condition `P` is:

(4)

```
(∃k)( P( [b ]*( k, s )) and
not [t]( [b]*( k, s )) and
(∀x)( 0 <= x < k => [t]( [b]*( x, s ))))
```

Antoy proves the equivalence of (4) and Dijkstra's definition, (1).

The pleasant surprise of Antoy's work is that once the functional abstraction of the body and test of the loop are specified (usually by a human), the weakest pre-condition of the loop is expressed as the above first order logical expression. Even if that formula is cumbersome for humans to manipulate, mechanical manipulation is not difficult [Antoy87].

Therefore, given (2) and a functional abstraction of the loop test, (4) is a first order expression of the weakest precondition of the loop for post-condition `P`.

This concludes our review of program verification formalisms.

4 Data Flow Testing for Array Elements

Before we introduce the wp_du method, we need to modify the data flow criteria we introduced in the last chapter such that these criteria can be used to test array elements.

To begin with, let's visualize what it means to employ data flow testing to investigate array elements. In our previous discussions of data flow testing, we considered each variable in the program and looked for paths which investigated the definition and use of that variable. To apply data flow testing to an array element, means to execute the program with input data which investigate the definition and use of that element. We will discover that this is more complicated than data flow testing for simple variables.

This chapter has three sections: In the first, we discuss why the data flow criteria we have defined so far cannot be used to test array elements. We observe that the expressiveness of graph theory is insufficient for the purpose of defining data flow criteria for array elements. Thus, we abandon graph theory. The second section redefines the previous data flow criteria without graph theory. In the last section, we define the data flow criteria for array elements.

Data Flow Criteria Do Not Test Array Elements

The reason our previous definitions of feasible data flow cannot easily be applied to array elements is because they do not make allowances for ambiguous array elements. An *ambiguous array element* is an array element reference where the expression which computes the subscript contains at least one variable. For example, if `a` is an array, then `a[ i ]` and `a[ x + y - 7 ]` are ambiguous array elements, while `a[ 3 ]` and `a[ 2 + 5 ]` are not. Ambiguous array elements pose many difficulties.

To illustrate these difficulties, suppose the following two statements occur in the middle of a complicated program:

```
a[ i ] := 0;
writeln( a[ j ] );
```

The difficulties are: First of all, we cannot always determine which array element is defined in the first statement or used in the second. Further, the array elements which actually are defined and used may depend on the input to the program. Finally, if these statements are executed more than once, they may define and use different array elements each time. Note that none of these difficulties exist with simple variables.

The result of these difficulties is that when we employ the existing data flow criteria, either original or feasible, to test a particular array element, for example `a[ 2 ]`, we will not always be able to determine what nodes and edges define and use `a[ 2 ]`, and further, even if we are able to do that, executing a path which includes that definition and that use does not ensure that `a[ 2 ]` will be defined or used during the execution of that path. Let's consider an example. Assume that `a` is an array indexed from `1` to `3`.

```
i := 1;
while i <= 3 do begin
     a[ i ] := i;
     i := i + 1;
end;

readln( i );
writeln( a[ i ] );
```

Figure 4.1

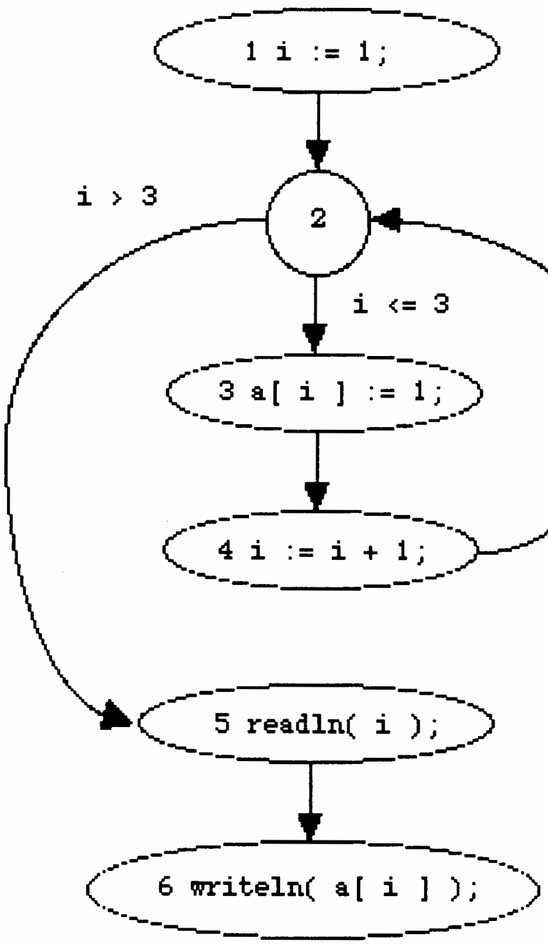

Figure 4.2

Let's assume that we are applying some feasible data flow criterion and we are particularly interested in investigating `a[ 2 ]`. In such a simple program, we can see that `a[ 2 ]` is defined at node `3` and, when the input datum read at node `5` is `2`, used at node `6`. Note that in the general case, due to unsolvability, we cannot always carry out such an identification. In this case, we consider ourselves lucky and proceed. The criterion requires that we find a feasible path which includes node `2` and later, before `a[ 2 ]` is defined again, includes node `6`. `( 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 5, 6 )` is such a path. Observe, however, that the criterion does not place any constraints on the input datum which the tester provides. Rather, the criterion only requires that the prescribed path be exercised. Therefore, the criterion is satisfied by executing the above path with an input datum which is not `2`. But we can see the test will only be useful if the input datum is `2`. Thus, the criteria as they are currently defined, cannot be used to test array elements.

The question we now consider is: can the current data flow criteria be extended to test array elements? In what remains of this section, we demonstrate that the answer is: not without extending or abandoning graph theory as the formalism in which we define the criteria.

Before we proceed, we need to establish a convention for discussing specific array elements, without the obligation of actually naming an exact array element for each discussion. We will let `a[ I ]` stand for some specific, unspecified array element. In more technical terms, `a` is a free variable ranging over the arrays in the program we are considering and `I` is a free variable which ranges over the set of valid indices for the array we select for investigation. We will use these free variables throughout this thesis.)

Note that `a[ I ]` is *not* an ambiguous array element, because we are using `I` to stand for a particular valid array index. We now turn to the inadequacies of graph theory as a formalism in which to define data flow testing for array elements.

We saw in the last example that one significant problem in trying to employ data flow criteria to test array elements is this: Once we have identified an array element `a[ I ]` we want to investigate, and a definition of `a[ I ]` called `DEF` and use of `a[ I ]` called `USE` (either of which may contain ambiguous array elements) which we want to test, simply exercising a path which contains `DEF` and `USE` is not sufficient to ensure that `a[ I ]` is defined at `DEF` or used at `USE`. Note that for `a[ I ]` to be truly tested with respect to `DEF` and `USE` requires executing the program with an input datum which exercises a path with the property that `a[ I ]` is defined at `DEF` and before `a[ I ]` is defined again, `a[ I ]` is used at `USE`. However, when we construct a set of paths to satisfy some data flow criterion, we have no opportunity to express this requirement. This is the problem.

To solve this problem, we could extend the data flow criteria the following manner: instead of constructing only a set of paths to satisfy a data flow criterion, we could instead construct a set of duples, where each duple contained a path and a def-use pair which that path must test. Although this is a solution to the above problem, we chose not to pursue it for reasons explained in the next section.

In summary, we have seen that the current data flow criteria cannot be used to test array elements.

Data Flow Criteria Expressed Without Graph Theory

In the previous section, we encountered problems in extending the data flow criteria to array elements. We traced those problems back to our use of graph theory. We now observe another reason that graph theory is inconvenient for our purposes.

The wp_du method, which we introduce in the next chapter, is built on Dijkstra's weakest pre-condition formalism [Dijkstra76], which makes no use of graph theory. Since it is easier to express the data flow criteria without graph theory than to express the weakest pre-condition formalism within graph theory, we here carry out the former.

For both of the aforementioned reasons, we redefine the data flow criteria we have encountered so far. More specifically, in this section, we express the data flow criteria in terms of the statements in the program, not the program graph.

We accomplish this redefintion of the criteria with an elegant procedure (i.e. a simple trick): Recall that all our data flow criteria have been defined using the terms: "definition of a variable", "use of a variable", "node", and "edge". If we simply redefine the first two terms so that they refer to the program itself, instead of the program graph, and go back to the data flow criteria definitions and replace all instances of "node" and "edge" by "statement", we will have effectively redefined the data flow criteria in the desired manner. That is how the goal of this section is met.

For example, a *definition* of a variable, which formerly was associated with a node in the program graph, is associated with the statement in the program which contains that definition. Similarly, a *c-use*, which was formerly associated with a node in the program graph, is associated with the statement in the program which contains that use.

Identifying *p-uses* with parts of the actual program is more complicated. Recall that a p-use of a variable was formerly an edge which departs from an empty node, where that empty node represents the beginning of an **if** or **while** statement which is controlled by a boolean expression which contains the variable we are considering. However, no edges actually exist in the program itself. So, for each variable in each boolean expression which controls an **if** or **while** statement, we associate two p-uses: one p-use of that variable is the statement which is executed immediately after the boolean expression evaluates to `true`; the other p-use of that variable is the statement which is executed immediately after the boolean expression evaluates to `false`. Thus, although it appears strange at first, for a given boolean expression which controls an **if** or **while** statement, each variable which occurs in that expression has two p-uses, and these p-uses are identified with the statements which follow the evaluation of that expression.

So, for example, returning to Figure 2.2 in Chapter two, the p-uses of x which were represented as:

```
USE: x, (2, 3)
USE: x, (2, 6)
USE: x, (3, 4)
USE: x, (3, 5)
```

when we were using graph theory, are now represented as:

```
USE: x, 3
USE: x, 6
USE: x, 4
USE: x, 5
```

A *use* is either a p-use or a c-use.

We have succeeded in associating the definitions of variables and uses with statements in the program itself. Since all our previous definitions depended on the definitions of "definition of a variable" and "use of a variable" the old data flow criteria are effectively redefined.

Now we can discuss applying data flow criteria to a program without reference to graph theory.

The Data Flow Criteria Extended to Test Array Elements

In this section, we define data flow criteria which test array elements. However, in contrast to how data flow criteria have been defined previously, we do not make use of graph theory, for reasons discussed in the earlier parts of this chapter.

Recall that the previous data flow criteria accept a set of paths as input and either accept or reject that set. The new criteria we define will accept as input a set of logical expressions and will either accept or reject the set.

Before we present the definitions of the criteria, we present two auxiliary definitions and a caveat.

We say that a statement *may define* an array element `a[ I ]` if that statement defines an ambiguous array element. A statement *may use* an array element `a[ I ]` if that statement uses an ambiguous array element.

We also define the locution: *a logical expression tests a def-use pair* *< DEF, USE, a[I]>* if and only if any input datum which satisfies that logical expression causes an execution of the program where a[I] is defined at DEF and then, before it is defined at another statement, used at USE. Note that if there is no such input datum for the specified def-use pair, then the logical expression is always false. We will see that the wp_du method gives a computable procedure for generating such logical expressions.

A final note of caution before we present the criteria definitions: The definitions we propose assume that the tester can carry out a task which is in general unsolvable. This is not scandalous: recall that the definitions of the feasible data flow criteria presented by Frankl and Weyuker require that the tester identify feasible and infeasible paths, which we know to be an unsolvable problem. The task we require is that the tester be able to determine whether a logical expression, perhaps in higher order logic, is equivalent to false or not. We will soon see why this task is necessary in our definitions of the data flow criteria which can test array elements.

We present the new definitions for all-defs*, all-p-uses/some-c-uses*, and all-uses* for testing array elements. For clarity, we present these definitions in two forms: in algorithmic and declarative forms. First we give the algorithmic definitions.

Each algorithm begins with an empty set of logical expressions L.

If we are applying the **all-defs*** criterion for array elements, then for each array element a[I]:

- For each definition called DEF which may define a[I] and for each use called USE which may use a[I], do the following:
    - Construct a logical expression that tests the def-use pair < DEF, USE, a[I] >.
    - If that expression is not logically equivalent to false, then add it to the set of expressions L.

Any set of logical expressions L constructed in this way satisfies the all-defs* data flow testing criterion for array elements.

If we are applying the **all-p-uses/some-c-uses criterion*** for array elements, then for each array element a[I]:

- For each definition called DEF which may define a[I], do the following:
    - If there exists a p-use called P_USE for which there exists a logical expression which tests < DEF, P_USE, a[I] > which is not equivalent to false then:

    - For every p-use called USE, construct a logical expression which tests < DEF, USE, a[I] >. Add each expression which is not equivalent to false to L.

  - If there are no such p-uses, then identify a c-use called C_USE for which there exists a logical expression which tests < DEF, C_USE, a[I] > which is not equivalent to false. Add that expression to L.

Any set of logical expressions L generated by following the above algorithm satisfies the all-p-uses/some-c-uses* data flow testing criterion for array elements.

If we are applying the **all-uses*** criterion for array elements, then for each array element a[I]:

- For each definition called DEF which may define a[I], do the following:

  - Identify a use called USE for which there exists a logical expression which tests < DEF, USE, a[I] > which is not equivalent to false. Add that expression to L.

Any set of logical expressions L generated by following the above algorithm satisfies the all-uses* data flow testing criterion.

We now present the declarative equivalents of the above algorithmic definitions.

Let L be a set of logical expressions.

L satisfies **all-defs*** if and only if:

```
for all arrays a and all array elements a[ I ] ,
        for all definitions DEF which may define a[ I ] ,
                for all uses USE which may use a[ I ] ,
                        E is a logical expression which tests
                        < DEF, USE, a[ I ] > and E is not false implies that
                         an expression which is logically equivalent to E is in L
```

To define all-p-uses/some-c-uses* we need the following definition: Let DEF be a statement which may define a[I], then:

```
dpu_arr( DEF, a[ I ] ) = { every p-use P_USE for which
   there exists a logical expression which tests < DEF,
   P_USE, a[ I ] > which is not equivalent to false }
```

L satisfies **all-p-uses/some-c-uses*** if and only if:

```
for all arrays a and all array elements a[ I ] ,
    for all definitions DEF which may define a[ I ] ,
        dpu_arr( DEF, a[ I ] ) is not empty implies
            for all p-uses P_USE which may use a[ I ],
                E is a logical expression which tests
                < DEF, P_USE, a[ I ] > and E is not false
                implies that an expression which is logically
                equivalent to E is in L
        and
        dpu_arr( DEF, a[ I ] ) is empty implies
            there exists a c-use C_USE which may use a[ I ] such that
                E is a logical expression which tests
                < DEF, C_USE, a[ I ] > and E is not false and
                an expression which is logically equivalent to
                E is in L
```

L satisfies **all-uses*** if and only if:

```
for all arrays a and all array elements a[ I ] ,
    for all definitions DEF which may define a[ I ] ,
        for all uses USE which may define a[ I ] such that
            E is a logical expression which tests
            < DEF, USE, a[ I ] > and E is not false and
            an expression which is logically equivalent to E is in L
```

5 The wp_du Method

In the previous chapters, we presented the background to the problem and layed the foundation for its solution. Recall the problem is that data flow testing cannot be used to test individual array elements. In this chapter, we explain the solution: the wp_du method. This chapter has two sections which explain how and why the wp_du method works, respectively. In the next chapter, we apply the wp_du method to example programs.

What the wp_du Method Does

In this section, we present a black box description of the wp_du method and a discussion of how repeated applications of the wp_du method can be used to characterize test sets which satisfy any specified data flow criterion. We defer an explanation of the mechanics of how the wp_du method actually works until the next section.

A Functional Description of the wp_du Method

The wp_du method requires three items as input:

1. a program, which is written in a structured, imperative language. For this thesis, we assume the program is written in the subset of Pascal described in the Introduction.

2. an indication from the tester of which variable we want to investigate in this application of the wp_du method. This variable can be either simple or an array element. Since the difficult case is the investigation of the array elements, we will assume that the variable we investigate is an array element. Let `a` stand for the array under inspection. Let `I` be a free variable which stands for the index of the array element we intend to investigate. So, if the tester wanted to investigate the third element of array `a`, for example, then `I` would stand for `3` for the duration of that application of the wp_du method.

3. an indication from the tester of a statement of definition, which we will identify by the name `DEF`, and a statement of use, which we will identify by the name `USE`. We assume that `DEF` defines some array element, possibly ambiguous, and `USE` uses some array element, possibly ambiguous.

The result of an application of the wp_du method is a logical expression which characterizes the input data which cause `DEF` to execute at least once such that `a[ I ]` is defined, and later, `USE` is executed at least once such that `a[ I ]` is used. Further, `a[ I ]` is not defined between the aforementioned executions of `DEF` and `USE`. In the language of the last chapter, the wp_du method constructs a logical expression which tests the def-use pair `< DEF, USE, a[ I ] >`.

Two comments: First, the reader who favors formalization may find the preceding description unsatisfying. Note that a formal treatment of the above would require the construction of an operational semantics which allows us to formally discuss sequences of states and statements [LoeSie87]. Such an undertaking is beyond the scope of the present work. Second, the logical expression produced by the wp_du method can be used in the construction of a test set in at least two ways: either by incorporating that expression into a larger logical expression which characterizes the entire test set or, using the expression to generate individual test data which are added to the test set.

wp_du and the Characterization of Test Sets which Satisfy A Specified Data Flow Criterion

With our general understanding of the wp_du method, we can substantiate the claim that repeated applications of the method can be used to construct logical expressions that characterize all test sets which satisfy any data flow criterion, feasible or not.

Of course, the actual construction of a test set which satisfies a specified data flow criterion is an unsolvable problem, so we can't expect our construction to succeed in every case. Our success in overcoming unsolvability will depend on our method for generating assignments which satisfy a given logical expression, which we do not consider here.

Suppose we choose a data flow criterion. For each variable, identify all the statements which may define that variable and all the statements which may use that variable. Note that the cartesian product of these definitions and uses is the set of all def-use pairs which may define and may use that variable.

Each data flow criterion requires that a certain number of these def-use pairs be tested. Regardless of what criterion is applied, each def-use pair can be passed as input to the wp_du method. The method will return a logical expression which characterizes all the input data which test that def-use pair.

If we apply the wp_du method to each def-use pair which our chosen criterion requires us to investigate, we will construct a logical expression which tests each def-use pair. To construct a test set which satisfies that criterion then requires that, for each logical expression, we generate a datum which satisfies that expression and add that datum to the test set. When we execute that program on all the data in the test set, we will have successfully tested the program with respect to the chosen criterion.

How the wp_du Method Works

In this section, we explain how the wp_du method works.

Recall from the previous section that the wp_du method takes three items as input: a program, a variable, and a pair of statements, one called `DEF`, which defines the variable, and one called `USE`, which uses the variable. (Recall that we are most interested in the case where this variable is an array element.) From this input, the method produces a logical expression which characterizes the input data which cause `DEF` to be executed such that the variable is defined, and later, `USE` is executed such that the variable is used, and finally, the variable is not defined between the aforementioned executions.

The wp_du method carries out this construction in two steps:

The **first step** is to construct a modified version of the original program. Call the original program `P` and the modified program `P'`. (Note that we usually call `P'` an "instrumented" version of `P`.)

`P'` is constructed such that it carries out the same computation as `P` except `P'` also contains code which implements a finite state machine which monitors whether the specified variable has been defined at `DEF` and not redefined before a use at `USE`. The finite state machine requires one variable called `status`, which can take on any of the following three values: `not_defined`, `defined`, and `def_used`. At the beginning of `P'`, `status` is initialized to `not_defined`. Further, `P'` is constructed such that `status` is set to `def_used` exactly during the executions where the specified variable has been defined at `DEF` and not redefined before a use at `USE`.

We will examine the finite state machine and the construction of `P'` soon.

The **second step** of the wp_du method is to compute the weakest pre-condition such that `P'` terminates in any state which satisfies `status = def_used`. This weakest pre-condition computation can be carried out in the same way as any other weakest pre-condition computation. We'll see examples of such computations in the next chapter. The result of this weakest pre-condition computation is a logical expression which characterizes all the input states for which `DEF` was executed such that the variable was defined, and after that, `USE` was executed such that the variable was used, and finally, that variable was not defined between the aforementioned executions.

Note that we do not use the weakest pre-condition formalism to prove the correctness of the programs we consider because if we were able prove the correctness of the program, then theoretically, we could dispense with testing.

Let's turn to the construction of the instrumented program mentioned in the first step.

We are given:

- a program `P`
- an indication of which array we are investigating, call it `a`
- an assignment to the free variable `I` which determines which array element we are investigating
- a statement `DEF` where `a[ I ]` may be defined, and
- a statement `USE` where `a[ I ]` may be used.

Note that in any actual application of the wp_du method, the I would be replaced by some value which specifies an array index. However, since we are discussing the wp_du method in the abstract, we will not assign a specific value to I.

We need to define a few more symbols. Let e_{DEF} represent the expression which computes the index of the array element defined in DEF. Let e_{USE} represent the expression which computes the index of the array element used in USE. We will use MORE_DEF to refer to a statement, which is not DEF, where a[I] may be defined. Each time we use the MORE_DEF symbol, we need to specify what statement it refers to. Once the denotation of MORE_DEF has been fixed, let e_{MORE_DEF} refer to the expression which computes the index which is defined by the MORE_DEF statement.

We transform P into P' by adding code which implements the aforementioned finite state machine. This is done in the following way.

Declare the variable status initialize it to not_defined.

Include the following code immediately before DEF:

```
if ( eDEF = I ) and ( status = not_defined ) then
     status := defined;
```

Include the following code immediately before USE:

```
if ( eUSE = I ) and ( status = defined ) then
     status := def_used;
```

Finally, identify all statements where a[I] may be defined, excluding DEF. We precede each such statement with the following code, where MORE_DEF refers to the statement currently being considered:

```
if ( eMORE_DEF = I ) and ( status = defined ) then
     status := not_defined;
```

Recall that I, e_{DEF}, e_{USE}, and e_{MORE_DEF} are free variables which do not appear in the actual text of P'. Rather, an index will be substituted for I and expressions will be substituted in for e_{DEF}, e_{USE}, and for each instance of e_{MORE_DEF}.

This code implements the following finite state machine where the state of the machine is the current value in the status variable:

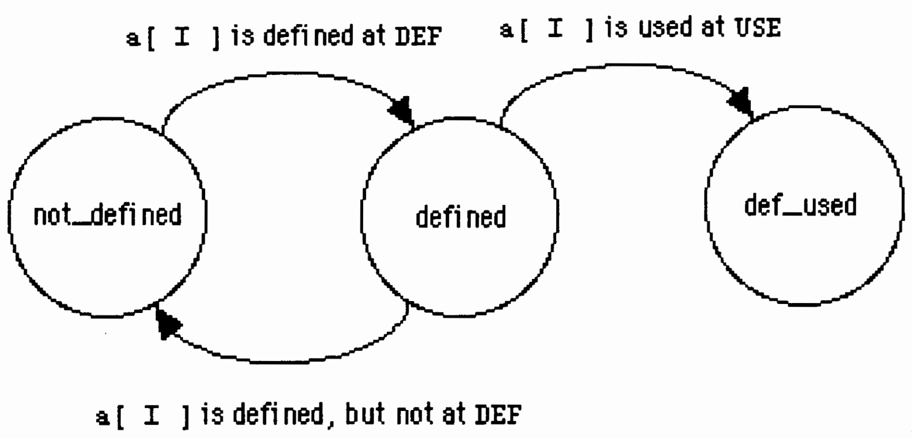

Figure 5.1

Note that:

- the edge which leaves the `not_defined` node represents the code we included before the `DEF` statement,
- the edge which connects the `defined` and `def_used` node corresponds to the code which we included before the `USE` statement, and
- the edge which connects the `defined` and `not_defined` node corresponds to the code we included before each statement, except `DEF`, which may define `a[ I ]`.

Since `status` is initialized to `not_defined`, `not_defined` is the start state. The finite state machine changes its state whenever the flow of control reaches one of the **if** statements we added and the the boolean statement which controls that **if** statement evaluates to `true`. In any other case, the finite state machine does not change its state. What state the program terminates in depends on the input state.

If the finite state machine is in the `not_defined` state (the variable `status` has the value `not_defined`), we know one of two things has occurred: Either `a[ I ]` has not been defined at `DEF` or else `a[ I ]` was defined at `DEF` but was subsequently defined at some other statement before being used at `USE`.

If the finite state machine is in the `defined` state, we know the most recent definition of `a[ I ]` occurred at `DEF`. (Note that `a[ I ]` can be defined at `DEF` many times in succession, so long as `a[ I ]` is not defined at some other statement besides `DEF`.)

If the finite state machine is in the `def_used` state, we know that `a[ I ]` was defined at `DEF` and, before it was defined at another statement, `a[ I ]` was used at `USE`. Note that once the finite state machine enters the `def_used` state, it cannot leave that state. The reason for this is that all that is required for an execution to test the def-use pair wrt a[I] composed of DEF and USE is for a[I] to be defined at DEF at some point in the execution, and then be used at USE before being defined at another statement.

Having completed the explanation of *how* the wp_du method works, we would like to consider *why* the method works. Note that a formal soundness proof of our method would require an operational semantics which is outside the scope of this thesis.

In place of a formal soundness proof, we will explain two relations which hold between the original program, P, and the corresponding instrumented program, P'. We assume that it is obvious that if these two relations hold, then the wp_du method is sound.

The first relation: for a given input state, P and P' always produce the same result, excluding the status variable. (In terms of program transformations [Morgan90], we say P' is a refinement of P.) This means, in addition, that P and P' terminate for the same input states.

To see that this relation holds between P and P', the reader must see that the **if** statements which are added to P to create P' do not alter the computation which the program carries out in any essential way. (Reminder: the status variable is used only in the **if** statements explained above, therefore, it does not appear anywhere else in P'.)

The second relation which holds between P and P' is the following:

Suppose we have specified an array element a[I] to investigate, and statements DEF and USE.

For all input state i for which P terminates,

> P executed with i causes DEF to be executed such that a[I] is defined, and before a[I] is defined again, USE is executed such that a[I] is used

if and only if

> P' executed with i causes P' to terminate with status containing the value def_used.

To see this relation holds between P and P', the reader should review the construction of the finite state machine which P' contains.

In conclusion, we have presented the wp_du method and a discussion of why it works. In the next chapter, we examine some examples of the wp_du method.

6 Examples of the wp_du Method

In the previous chapter, we presented the wp_du method and discussed how repeated applications of the method to different def-use pairs can characterize all the test sets which satisfy any specified feasible data flow criterion and program. In this chapter, we present two programs to which we apply the wp_du method. For each program, we identify one def-use pair with respect to a particular array element `a[ I ]`, and compute the logical expression which characterizes all the input data which test that def-use pair.

The programs we present are simple for two reasons: 1) we want the reader to be able to intuitively verify that the wp_du method produces the correct result and 2) the weakest pre-condition computations for even moderately complicated programs are intractable for humans. Mechanical assistance is clearly required for applying the wp_du method to such programs.

In the first example, we work through the weakest pre-condition computation and supporting proofs in some detail. In the second example, we omit most of these details.

All of our examples use an array `a` with elements indexed from `1` to `n`.

Example: Squaring

Consider the following program which assigns to each array element the square of its index. The algorithm used by this program is based on the fact that the square of a number `x` is the sum of the odd integers from `1` to `2 * x - 1`.

```
a[ 1 ] := 1;
i := 2;

while i <= n do begin
     a[ i ] := a[ i - 1 ] + 2 * i - 1;
     i := i + 1
end;
```

Figure 6.1

The instrumented version of this program is:

```
status := not_defined;
a[ 1 ] := 1;
i := 2;

while i <= n do begin
     if ( I = i - 1 ) and ( status = defined ) then
          status := def_used;
     if ( I = i ) and ( status = not_defined ) then
          status := defined;
     a[ i ] := a[ i - 1 ] + 2 * i - 1;
     i := i + 1
end;
```

Figure 6.2

Excluding the definition of a[1] at the beginning of the program, there is only one definition of an array element and one use of an array element. Notice that both occur in the same statement. Therefore, we identify both DEF and USE with this statement. Consequently, we precede DEF and USE statement with the code which monitors the definitions and uses performed by this statement: a[i] := a[i - 1] + 2 * i - 1;. The order of these if statements is of no consequence. Finally, recall that there is no other statement besides a[1] := 1; which defines an array element. Since we can see that the only array element that is defined by this statement is a[1], we omit the MORE_DEF code which monitors this statement.

Before we compute the weakest pre-condition of this code with respect to the post-condition status = def_used, it is worthwhile to anticipate what result we expect to achieve. Our computation will yield a weakest pre-condition which characterizes all the input data which causes a[I] to be defined at DEF and then used at USE without being defined any where else in between. (Recall that I is a free variable ranging over the set of valid indices for array a.)

Usually, when the wp_du method is applied, I is set to the value of a particular index. However, for simple programs like this one, we leave I unspecified, and derive results that apply to all array elements.

Inspection of the code reveals that an array element a[I] will be defined at DEF and used at USE, without intermediate definition, in the case where I is greater than 1 and less than n. Said in a different way, a[1] is never defined at DEF and a[n] will never be used at USE. But if I is given any value between these two extremes, then any execution of the program will test this def-use pair. We see now that the weakest pre-condition produces this result.

Our proof strategy is to display a sequence of predicates { G_n } which we prove, via an induction argument, to be equivalent to the sequence of predicates { H_n(status = def_used) } which define the weakest pre-conditions for loop iterations. Then the existentially quantified k in Dijkstra's definition of the weakest precondition of a while statement will be instantiated with the number of loop iterations. The resulting logical expression will not contain any predicate transformers, and so will be (relatively) easy to manipulate. We will pass the resulting logical expression through the predicate transformers of the assignment statements which precede the loop and the result described in the previous paragraph will be obtained.

Define the sequence of functions { G_k } for all natural numbers k:

```
G0 = H0( status = def_used )

for all natural numbers k > 0,
Gk =
        i + k - 1 = n and
        (( I = i - 1 and status <> not_defined ) or
        I in [ i..i + k - 2 ] or
        status = def_used )
```

Note that in is being used to mean "is a member of" and [a..b] is the set of all integers between a and b inclusive.

Before we proceed with the proof, let's try to interpret the meaning of G_k for an unspecified k greater than 0. Recall that G_k is meant to be the weakest precondition such that the loop iterates exactly k times and then terminates in any state satisfying status = def_used. G_k is a logical conjunction. The first conjunct, i + k - 1 = n, characterizes the states which will cause the loop to iterate k times and then terminate. The second conjunct characterizes the states which cause the loop to terminate in a state satisfying status = def_used. This second conjunct is composed of three disjuncts, which we examine.

If the loop begins in a state where I = i - 1 and status <> not_defined is true, then, in the first iteration, a[I] will be used at USE and status will be set to def_used, if it does not already contain that value. If the loop begins in a state where I in [i..i + k - 2] is true, then regardless of the value of status, a[I] will be defined at DEF and used at USE without intermediate definitions during the k iterations of the loop. Finally, if the loop begins in a state where status = def_used is true, then, of course, it will terminate in a state satisfying status = def_used.

We will begin our induction proof by establishing the base case that G_1 = H_1(status = def_used) and then take the inductive step where we assume that for some natural number k, G_k = H_k(status = def_used) and then we prove that G_{k+1} = H_{k+1}(status = def_used).

Base Case

By definition,

```
G1 =
i = n and
(( I = i - 1 and status <> not_defined ) or
status = def_used )
```

To see that `G1` and `H1( status = def_used )` are equal, we need to represent `H1( status = def_used )` without `wp` predicates. This can be done by evaluating the predicate transformers.

```
H1( status = def_used ) =

i <= n and
wp( loop body, H0( status = def_used )) =

i <= n and
wp( first if; second if;
     a[ i ] := a[ i - 1 ] + 2*i - 1; i := i + 1,
          i > n and status = def_used ) =

i <= n and
wp( first if; second if; a[ i ] := a[ i - 1 ] + 2*i - 1,
     i + 1 > n and status = def_used )
```

We now need to evaluate the above predicate transformer. This will be done in two steps. First, we will now compute:

```
wp( second if; a[ i ] := a[ i - 1 ] + 2 * i - 1,
     i + 1 > n and status = def_used )
```

and call the result `WP_IF2`. Then we will compute

```
wp( first if, WP_IF2 )
```

and call the result `WP_IF1`. We will then see that

```
i <= n and WP_IF1
```

which will be equal to `G1`. Step one:

```
wp( second if; a[ i ] := a[ i - 1 ] + 2 * i - 1,
     i + 1 > n and status = def_used ) =

(( I = i and status = not_defined ) =>
     ( i + 1 > n and defined = def_used )) and
(( I <> i or status <> not_defined ) =>
     ( i + 1 > n and status = def_used )) =
i + 1 > n and
( I <> i or status <> not_defined ) and
(( I <> i or status <> not_defined ) =>
     status = def_used ) =

i + 1 > n and
( I <> i or status <> not_defined ) and
status = def_used
```

which we call `WP_IF2`.

Step two:

```
wp( first if, WP_IF2 ) =

i + 1 > n and
(( I = i - 1 and status = defined ) =>
     (( I <> i or def_used <> not_defined ) and
          def_used = def_used )) and
(( I <> i - 1 or status <> defined ) =>
     (( I <> i or status <> not_defined ) and
          status = def_used )) =

i + 1 > n and
(( I <> i - 1 or status <> defined ) =>
     (( I <> i or status <> not_defined ) and
          status = def_used )) =

i + 1 > n and
(( I = i - 1 and status = defined ) or
(( I <> i or status <> not_defined ) and
     status = def_used ))
```

Call this `WP_IF1`. Note that

```
H1( status = def_used ) =
i <= n and WP_IF1
```

which is equal to:

```
i = n and
(( I = i - 1 and status = defined ) or
(( I <> i or status <> not_defined ) and
     status = def_used ))
```

To see that this representation of H_1`( status = def_used )` is equivalent to G_1, we need to do a substantial amount of symbol manipulation which involves distributing `and` and `or`. Here it is:

```
i = n and
(( I = i - 1 and status = defined ) or
(( I <> i or status <> not_defined ) and
     status = def_used )) =

i = n and
(( I = i - 1 and status = defined ) or
( I <> i and status = def_used ) or
( status <> not_defined and status = def_used )) =

i = n and
(( I = i - 1 and status = defined ) or
( I <> i and status = def_used ) or
status = def_used ) =

i = n and
(( I = i - 1 and status = defined ) or
status = def_used ) =

i = n and
(( I = i - 1 or status = def_used ) and
( status = defined or status = def_used )) =

i = n and
(( I = i - 1 or status = def_used ) and
status <> not_defined ) =

i = n and
(( I = i - 1 and status <> not_defined ) or
( status = def_used and status <> not_defined )) =

i = n and
(( I = i - 1 and status <> not_defined ) or
status = def_used )
```

which is equal to G_1.

Inductive Step

Now, we take the inductive step. Given that for some k, G_k = H_k(status = def_used), we need to prove that G_{k+1} = H_{k+1}(status = def_used).

We begin with the definition of H_{k+1}(status = def_used):

```
Hk+1( status = def_used ) =
i <= n and
wp( loop body, Hk( status = def_used ))
```

Let's focus on the wp(loop body, H_k(status = def_used)) conjunct.

```
wp( loop body, Hk( status = def_used )) =
wp( loop body, Gk )
```

by inductive hypothesis. We will substitute in the definition of `Gk` and show that `wp( loop body, Gk ) = Hk+1( status = def_used )`. Let:

```
wp( loop body, Gk ) =
wp( first_if; second_if;
     a[ i ] := a[ i - 1 ] + 2 * i -1; i := i + 1, Gk )
```

We now need to evaluate the above predicate transformer. We will do this is three steps. First, we will first compute:

```
wp( a[ i ] := a[ i - 1 ] + 2 * i -1; i := i + 1, Gk )
```

and call the result `WP_POST_IF2`. Second, we will compute:

```
wp( second_if, WP_POST_IF2 )
```

and call the result `WP_IF2`. In the third step, we will compute:

```
wp( first_if, WP_IF2 )
```

and call the result `WP_IF1`. Finally, we will see that

```
Gk+1 = i <= n and WP_IF1
```

We then conclude that `Gk+1 = Hk+1( status = def_used )`, which will complete our induction.

Step one:

```
wp( a[ i ] := a[ i - 1 ] + 2 * i -1; i := i + 1, Gk ) =

i + k = n and
(( I = i and status <> not_defined ) or
I in [ i + 1..i + k - 1 ] or
status = def_used )
```

Call the expression on the right hand side of the equality `WP_POST_IF2`. Step two:

```
wp( second_if, WP_POST_IF2 ) =

(( I = i and status = not_defined ) =>
     ( i + k = n and
     (( I = i and defined <> not_defined ) or
     I in [ i + 1..i + k - 1 ] or
     defined = def_used ))) and
(( I <> i or status <> not_defined ) =>
     ( i + k = n and
     (( I = i and status <> not_defined ) or
     I in [ i + 1..i + k - 1 ] or
     status = def_used )) =

i + k = n and
(( I = i and status = not_defined ) =>
     (( I = i and defined <> not_defined ) or
     I in [ i + 1..i + k - 1 ] or
     defined = def_used )) and
(( I <> i or status <> not_defined ) =>
     (( I = i and status <> not_defined ) or
     I in [ i + 1..i + k - 1 ] or
     status = def_used )) =

i + k = n and
(( I = i and status = not_defined ) =>
     ( I = i or
     I in [ i + 1..i + k - 1 ] or
     defined = def_used )) and
(( I <> i or status <> not_defined ) =>
     (( I = i and status <> not_defined ) or
     I in [ i + 1..i + k - 1 ] or
     status = def_used )) =

i + k = n and
(( I <> i or status <> not_defined ) =>
     (( I = i and status <> not_defined ) or
     ( I in [ i + 1..i + k - 1 ] or
     status = def_used )) =

i + k = n and
(( I = i and status = not_defined ) or
( I = i and status <> not_defined ) or
I in [ i + 1..i + k - 1 ] or
status = def_used ) =

i + k = n and
( I = i or I in [ i + 1..i + k - 1 ] or
status = def_used ) =

i + k = n and
( I in [ i..i + k - 1 ] or
status = def_used )
```

Call this `WP_IF2`. Step three:

```
wp( first_if, WP_IF2 ) =

i + k = n and
(( I = i - 1 and status = defined ) =>
     ( I in [ i..i + k - 1 ] or
     def_used = def_used )) and
(( I <> i - 1 or status <> defined ) =>
     ( I in [ i..i + k - 1 ] or
     status = def_used )) =

i + k = n and
(( I = i - 1 and status = defined ) or
I in [ i..i + k - 1 ] or
status = def_used )
```

By distributing `or` and `and` operators, we derive G_{k+1}. Here it is:

```
i + k = n and
(( I = i - 1 and status = defined ) or
I in [ i..i + k - 1 ] or
status = def_used ) =

i + k = n and
( I in [ i..i + k - 1 ] or
( I = i - 1 or status = def_used ) and
( status = defined or status = def_used )) =

i + k = n and
( I in [ i..i + k - 1 ] or
( I = i - 1 or status = def_used ) and
status <> not_defined ) =

i + k = n and
( I in [ i..i + k - 1 ] or
( I = i - 1 and status <> not_defined ) or
( status = def_used and status <> not_defined )) =

i + k = n and
( I in [ i..i + k - 1 ] or
( I = i - 1 and status <> not_defined ) or
status = def_used )
```

which is just G_{k+1}. Thus, the inductive step is complete. Thus we replace H_k`( status = def_used )` with G_k.

```
wp( status := not_defined; a[ i ] := 2; i := 2,
     there exists a k such that Gk ) =
2 + k = n and
( I in [ 2..k + 1 ] or
( I = 1 and not_defined <> not_defined ) or
not_defined = def_used ) =

2 + k = n and I in [ 2..k + 1 ]
```

Let `k` be instantiated with `n - 2` to receive:

```
I in [ 2..n - 1 ]
```

which is the predicate `1 < I < n`. And thus we get the result we anticipated.

Example: Reverse

The following program reverses the contents of an array. This program is interesting because no array element is defined and then later used. Rather, they are all used and then defined. In this sense, this program is a null case for the wp_du method. Since no array element is defined and then used, the wp_du method should compute a logical expression equivalent to `false`. This is what happens.

We arbitrarily selected the statement `a[ n + 1 - i ] := t;` to be `DEF` and the statement `t := a[ i ];` to be `USE`. Note that in addition to `DEF`, the statement `a[ i ] := a[ n + 1 - i ];` also defines an arbitrary array element. That statement, therefore, is `MORE_DEF`.

```
status := not_defined;
i := 1;

while i <= n div 2 do begin
     if ( i = I ) and ( status = defined ) then
          status := def_used;
     t := a[ i ];
     if ( i = I ) and ( status = defined ) then
          status := not_defined ;
     a[ i ] := a[ n + 1 - i ];
     if ( n + 1 - i = I ) and ( status = not_defined )
then
          status := defined;
     a[ n + 1 - i ] := t;
     i := i + 1
end;
```

Figure 6.2

Again, we use the ingenuous method. Below is the $\{ G_k \}$ sequence. We do not include the proof that it is equivalent to $\{ H_k(\text{status} = \text{def\_used}) \}$ for all `k`.

$G_0 = H_0($ status = def_used $)$

for all natural numbers k,
$G_{k+1} =$

i + k = n div 2 and

(i + k = I and status = defined) or

$$(\bigvee_{x=2}^{k+1} (((\bigwedge_{y=x}^{k+1} n + 2 - y - i <> I) \text{ or status } <> \text{ defined })$$

and (i + x - 2 = I and status = defined))) or

$$(((\bigwedge_{x=1}^{k+1} n + 2 - x - i <> I) \text{ or status } <> \text{ not\_defined })$$

and status = defined)

Given that the $\{ G_k \} = \{ H_k($ status = def_used $) \}$, we follow the same procedure as in the previous example: We substitute G_k for $H_k($ status = def_used $)$ in Dijkstra's definition of the weakest pre-condition for a loop. Then, we instantiate k to be the number of iterations, which in this case, is n div 2. Finally, we pass the resulting expression through the weakest pre-condition predicate transformers of the statements which precede the loop. The resulting logical expression evaluates to false.

In conclusion, we have seen the application of the wp_du method to simple examples.

7 Conclusion

Recall that the problem which motivated this thesis is that data flow testing could not be applied to array elements.

In this thesis, we accomplished the following:

- Summarized path testing, data flow testing, and program verification formalisms.

- Extended the data flow criteria to include the ability to test array elements. This involved abandoning graph theory as the formalism within which we defined our criteria, and then redefining those criteria in terms of sets of logical expressions.

- Invented and justified the wp_du method and worked through some examples.

The wp_du method is noteworthy because it offers a computable procedure for constructing a logical expression which tests any def-use pair. These logical expressions can be used to construct test sets which satisfy any chosen data flow criterion. That is, although unsolvability permeates data flow testing, the wp_du method avoids the limitations of unsolvability. Unsolvability looms only when we try to generate data which satisfy some logical expression.

Bibliography

[Antoy87] Antoy, S.: Automatically Provable Specifications, Ph.D. Dissertation, University of Maryland, UMIACS-TR-87-28, CS-TR-1876, 1987.

[Backh86] Backhouse, R. C.: Program Construction and Verification, Prentice-Hall, 1986.

[Church36] Church, A.: A Note on the Entscheidungsproblem, Journal of Symbolic Logic, vol. 1 no. 1, 1936.

[DerPla88] Dershowitz, N. and Plaisted, D. A.: Equational Programming, Machine Intelligence 11, Clarendon Press, 1988.

[Dijkstra72] Dijkstra, E. W.: Notes on Structured Programming, contained in Structured Programming, Academic Press, 1972.

[Dijkstra76] Dijkstra, E. W.: A Discipline of Programming, Prentice-Hall, 1976.

[Durbin92] Durbin, J. R.: Modern Algebra, John Wiley and Sons, 1992.

[FraWey88] Frankl, P. G. and Weyuker, E. J.: An Applicable Family of Data Flow Testing Criteria, IEEE Trans. Software Eng., vol. 14, pp. 1483-1498, Oct. 1988.

[Gries81] Gries, D.: The Science of Programming, Springer-Verlang, New York, 1981.

[Hamlet88] Hamlet, R.: Special Section on Software Testing, Comm. ACM, vol. 31, pp. 662-667, Jun. 1988.

[HamGN93] Hamlet, D., Gifford, B. and Nikolik, B.: Exploring Dataflow Testing of Arrays, Proceedings of 15th Int. Conf. on Software Eng, May, 1993.

[HamTay90] Hamlet, D. and Taylor, R.: Partition Testing Does Not Inspire Confidence, IEEE Trans. Software Eng., vol. 16, pp. 1402-1411, Dec. 1990.

[HanKin76] Hantler, S. L. and King, J. C.: An Introduction to Proving the Correctness of Programs, ACM Computing Surveys, Vol. 8, pp. 331-353, Sept. 1976.

[Hoare69] Hoare, C. A. R.: An Axiomatic Basis for Computing Programming, Comm. ACM, vol. 12, pp. 576-580, 583, Oct. 1969.

[Howden76] Howden, W. E.: Reliability of the Path Analysis Testing Strategy, IEEE Trans. Software Eng., vol. SE-2, pp. 37-44, Sept. 1976.

[LoeSie87] Loeckx, J. and Sieber, K.: The Foundations of Program Verification, John Wiley and Sons, 1987.

[ManGhe87] Mandrioli, D. and Ghezzi, C.: Theoretical Foundations of Computer Science, John Wiley and Sons, 1987.

[Morgan90] Morgan, C.: Programming from Specifications, Prentice-Hall, 1990.

[RaWey85] Rapps, S. and Weyuker, E. J.: Selecting software test data using data flow information, IEEE Trans. Software Eng., vol. SE-11, pp. 367-375, Apr. 1985.