Black & White Testing:
Bridging Black Box Testing and White Box Testing.

E. Steegmans, P. Bekaert, F. Devos, G. Delanote, N. Smeets, M. van Dooren and J. Boydens

Department of Computer Science, K.U.Leuven

Abstract

Since the mid 1970s, software testing has been dominated by two major paradigms, known as black box testing and white box testing. Strategies for black box testing are based on the specification of the software component under test. Test suites developed using the black box paradigm reveal errors against the functional requirements of software components. Strategies for white box testing are based on the internals of software components. White box test suites are derived from the source code of the software component under test. Because programs can be represented in terms of graphs, solid coverage criteria can be defined for white box testing strategies. In this paper, we start with a brief overview of the principles underlying black box testing and white box testing. Hereafter, we propose a process for testing software components in which both paradigms are used at appropriate points in the software life cycle. We also touch upon the applicability of both paradigms in testing object-oriented programs.

1. Introduction

Strategies for software testing have traditionally been classified in black box testing strategies and white box testing strategies. Black box testing starts from the specification of a software component, resulting in test suites that check the correctness the external behavior of the component under test. Black box testing has not been in use that much in the software industry, largely because proper documentation is often lacking for software components. The introduction of object-oriented programming, amplified by an increased interest in pluggable software components is slowly changing this (bad) practice. Indeed, if software components are to be re-used over and over again, proper documentation is a fundamental requirement. This explains a renewed interest in strategies for black box testing.

White box testing builds upon the internals of a software component. Strategies for white box testing are not facing the problems of black box testing; a software component without an internal implementation simply cannot be deployed. Another big advantage of white box testing is its sound mathematical background. The source code of a software component can be transformed into a graph, meaning that all the mathematical machinery underlying graph theory can be applied in developing white box testing strategies. The most important contribution of these mathematical foundations is the definition of solid criteria for checking the coverage of a test suite. All this explains why white box testing has been in use for a number of decades. Recently, interest in the area of software testing has increased. This has resulted in a series of more advanced white box testing strategies such as dataflow testing and program slicing.

Strategies for software testing are applied to software systems and their components. For both black box testing and white box testing, the *software components* under test are routines. A routine can range from a very simple function to a complete program. Imperative programming languages use different terms for routines, such as *functions* and *procedures* in older generations of programming languages, and *methods* in object-oriented programming languages. The unifying characteristic of a routine is that it produces some outputs, given some inputs. Inputs can be supplied directly via a list of arguments, or indirectly by means of some stored data. In the same way, outputs can be yielded directly using some return types or indirectly by changing the state of some stored data.

In this paper, we start in section 2 with a survey of the paradigm of black box testing. We discuss the basic principles underlying black box testing, illustrate with some example strategies and conclude with pros and cons of black box testing. In section 3, we continue with an overview of white box testing. As for black box testing, we discuss the principles of this paradigm, we illustrate it with some strategies and discuss the strengths and weaknesses of this approach. In section 4, we illustrate an overall strategy in which black box testing and white box testing are combined. In this way, the strengths of both paradigms are combined, and their weaknesses are largely eliminated.

2. Black Box Testing

The paradigm of black box testing states that test suites must be derived from the *specification* or the *documentation* of the component under test. Strategies for black box testing do not use any information concerning the internals of the tested component. The specification of a component describes the outputs produced by the component for each possible set of input values. Specifications of software components are said to be *declarative*: they describe *what* can be expected from a component, without revealing *how* the component achieves its effects. For that reason, black box testing is often referred to as *functional testing*, because it is directed towards the external behavior of the tested component.

In this section, we first look at techniques to work out proper documentation for components. In section 2.1, we introduce the paradigm of *Design by Contract* for developing specifications of software components. In section 2.2 we continue with a brief overview of some of the most widespread strategies for developing black box tests. In section 2.3, these different strategies for black box testing are evaluated. Finally, we discuss the major pros and cons of black box testing in section 2.4.

2.1 Documentation of Software Components

Writing proper documentation for software components is often neglected at the different stages of the software life cycle. Much too often, the internals of software components are developed without an in-depth study of their external behavior. It is generally known that a lack of proper documentation is one of the main reasons for high maintenance costs. The introduction of object-oriented programming, building further upon the general notion of abstract data types, has given some new impulses in the area of software documentation. Indeed, one of the key concepts of object-oriented progamming is *encapsulation*, which states that implementation details of objects must be hidden from outside users.

An immediate consequence of encapsulation is that the behavior of a software component is not exposed in terms of its internal behavior. This means that other formalisms are needed to explain the behavior of a software component to its users. The paradigm of *Design by Contract* [Meyer] introduces some rather simple concepts to structure the documentation of software components in a rigorous way. Basically, the paradigm states that the documentation of a software component must be interpreted as a contract between its users on the one hand, and the developers of the component on the other hand. Both parties involved in the contract are assigned some rights and some duties. Clients of software components must obey all the conditions, known as preconditions, imposed on the usage of the component. Developers on the other hand, must produce all the effects, resulting from a proper usage of the component. Clients have the right to expect proper results; developers have the right to expect proper usage.

Strictly speaking, *Design by Contract* uses a formal language to specify preconditions, postconditions and class invariants. This formal language is typically some form of first-order logic intertwined with set-theoretical expressions. In practice, formal specifications are only rarely used in documenting software components. However, the ideas underlying Design by Contract can be applied just as well using some natural language to express preconditions, postconditions and class invariants. The ever growing interest for Model Driven Architecture[Frankel] may give new impulses to a more formal specification of software components during the early stages of software development. However, it is not to be expected that formal specifications will be used in the near future on large scales in practical software engineering.

Example 1 illustrates the basic principles underlying *Design by Contract* by means of a specification of a method to compute the greatest common divisor of two given integer numbers. The specification is worked out in Java[Eckel], using concepts offered by javadoc to structure the documentation in a number of sections. In Java, documentation for methods is structured using so-called tags.

- The tag `@param` is used to describe the role of each of the arguments in the specified method.
- The tag `@return` is used to describe the result returned by the specified method. In Example 1, two successive return-clauses specify that the value returned by the method is indeed the greatest common divisor of the given numbers. Notice that these clauses are specified both formally and informally. The formal specifications uses Java operators, extended with notations for quantifiers, such as the universal quantifier[1].
- The tag `@throws` is used to describe exceptions that can be thrown by the specified method. The method for computing the greatest common divisor may throw an Illegal Argument Exception, indicating that it was not possible to compute the specified result. According to the specification worked out in Example 1, the method may throw this exception if at least one of the given numbers is negative, or if both given numbers are zero.

```
/**
 * Compute the greatest common divisor of the given values.
 *
 *  @param   a,b
 *           The given integer numbers.
 *  @return  The resulting value is a common divisor of both
 *           given values, i.e. it divides both ‘a’ and ‘b’.
 *           | (a % result == 0) && (b % result == 0)
 *  @return  No larger integer value exists that divides
 *           both ‘a’ and ‘b’.
 *           | for each number in result+1..Long.MAX_VALUE:
 *           |    (a % number != 0) || (b % number != 0)
 *  @throws  IllegalArgumentException
 *           At least one of the given values is negative.
 *           | (a < 0) || (b < 0)
 *  @throws  IllegalArgumentException
 *           Both ‘a’ and ‘b’ are zero.
 *           | (a == 0) && (b == 0)
 */
public static long gcd (long a, long b)
    throws IllegalArgumentException
```

Example 1: Specification of a Java-method to compute the greatest common divisor.

Other tags are introduced or proposed to document Java programs. As an example, simple tags such as `@author` and `@see` are used to identify the author a some class definition or method, respectively to refer to other parts of the documentation. Other languages have introduced similar formalisms for documentation purposes. As an example, C# uses XML as a flexible notation for documenting classes and their members. As another example, the Unified Modeling Language (UML) [Fowler] fully supports preconditions, postconditions and invariants in developing class diagrams at the level of object oriented design. The Unified Modeling Language is complemented with the Object Constraint Language (OCL) [Warmer] to specify design elements in a formal way.

[1] Readers not familiar with formal specifications in logic, may skip them. A firm understanding of formal specifications is not needed to understand the rest of the paper.

2.2 Strategies for Black Box Testing

In this section, we briefly discuss two of the most popular strategies for black box testing. In section 2.2.1, we discuss boundary value testing as a simple approach for setting up a suite for testing the external behavior of a software component. Being a simple technique, boundary value testing can be automated to a large extent. Section 2.2.2 introduces equivalence class testing as a more in-depth strategy for black box testing. This approach requires more input from the test team and rewards this with a test suite of superior quality.

2.2.1 Boundary Value Testing

Boundary value testing is a simple strategy for setting up a series of tests for software components. The strategy starts from the observation that lots of errors tend to occur near extreme values for input variables. A study ordered by the U.S. Army revealed that a large portion of errors in software systems are boundary value faults. A typical example are loops iterating through a sequence of elements. Often such loops fail to handle the last element in the sequence, or attempt to iterate one more element beyond the last element in the sequence.

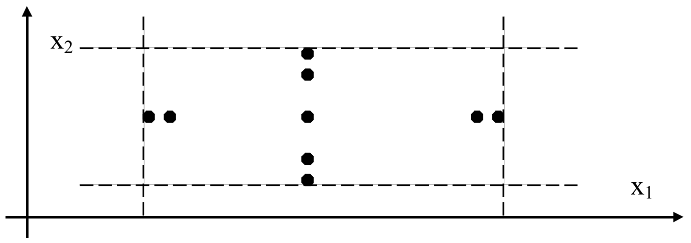

Figure 1: Principles of Boundary Value Testing.

The basic principles underlying boundary value testing are illustrated in Figure 1. For each input variable, tests will be set up involving (1) the minimum value of that input variable, (2) a value just above that minimum value, (3) a normal value, (4) a value just below the maximum value, and (5) the maximum value for that input variable. For the algorithm to compute the greatest common divisor, this strategy would result in testing the method with the following values for the first argument: `0`, `10`, `Long.MAX_VALUE/2`, `Long.MAX_VALUE-10` and `Long.MAX_VALUE`. For all these tests, some arbitrary value for the second argument will be used. Similar tests must be worked out covering the range of the second argument.

Several variants on the basic strategy for boundary value testing have been proposed. Robustness testing extends the tests generated by a basic boundary value testing strategy with values outside the regular domain of input values. In the example of computing the greatest common divisor, this would lead to extra tests for which the method has announced to throw exceptions. Another extension to boundary value testing is worst-case testing. In this strategy, all the selected values in the different ranges for input arguments are combined one by one. For the greatest common divisor, this strategy would for instance include a test involving the largest possible value for both arguments. Special value testing is yet another variation on the general theme of boundary value testing. In this strategy, the test suite is extended with tests covering special values for input variables. With this strategy, the test suite for the method computing the greatest common divisor might be extending with cases testing the correctness of the method with the value 0 for both arguments.

Boundary value testing is supported by several commercially available tools, generating test suites for methods involving arguments of primitive types. It is indeed not hard to see, that most of the work underlying boundary value testing can be automated. All a decent tool needs to find out is the possible range of values for the different arguments involved. The generation of the test suite is then a pure mechanical activity. An example of a tool supporting value boundary testing is the so-called *T Tool* which is integrated in several integrated development environments (IDE).

2.2.2 Equivalence Class Testing

Equivalence class testing [Myers, Mosley] is a more refined strategy for building black box tests. In this strategy, the domain of possible input values is partitioned into disjoint subsets. The basic criterion to partition the input domain is that the software component under test behaves the same for each set of values in a particular subset. An actual test is then included in the test suite for one representative set of values from each subset. For some methods, it is more appropriate to partition the domain of possible output values, instead of the domain of input values. In that case, the individual tests in the test suite will be such that they each generate an output in a different subset of the partition.

The basic principles underlying equivalence class testing are illustrated in Figure 2. In the example, the range of input values for the first argument is partitioned into three subsets; the range of input values for the second argument is partitioned into two subsets. In total, this results in a portioning of the input domain into 6 subsets. The generated test suite will use a representative value from each of these six subsets. For the algorithm to compute the greatest common divisor, there is no reason to partition the input domain. As a consequence, basic equivalence testing for this method would be restricted to a single test.

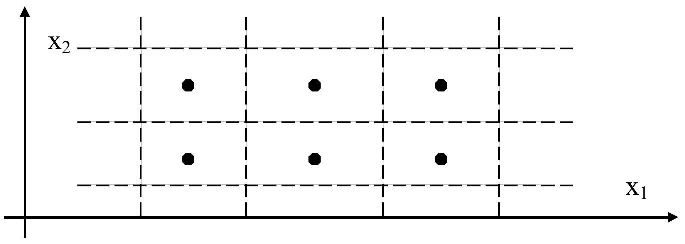

Figure 2: Principles of Equivalence Class Testing.

As for value boundary testing, several types of equivalence class testing have been proposed. First of all, one distinguishes between weak and strong strategies for equivalence class testing. The strategy described is referred to as strong equivalence testing: the test suite is build using a single representative from each subset resulting from the partitioning of the input domain. In weak equivalence testing, the test suite is constructed in such a way that a value from each partition is taken. In the example of Figure 2, three test cases would be sufficient in that case. A further portioning distinguishes between normal and robust equivalence class testing. Strategies for normal equivalence testing restrict their values to legal values for each of the input variables. In this respect, the strategy described in Figure 2 is a normal equivalence testing strategy. In a robust equivalence testing, the test suite is extended with illegal values for the different input variables.

It must be obvious that equivalence class testing has the potential to result in test suites of superior quality. First of all, there are good reasons to believe that the test suite is more complete. By definition, different cases that can be distinguished in the specification of a method will be covered by different tests. This is not at all guaranteed by value boundary testing. On top of more complete test suites, strategies for equivalence class testing will avoid redundant tests. This is one of the major problems of boundary value testing strategies, which will typically produce lots of tests, which all cover more or less the same case in the specification of the software component under test.

Tools supporting strategies for equivalence class testing are harder to develop. The major problem is derive equivalence classes from method specifications. So far, only some experimental tools are available. Some difficult problems must still be resolved before they can really be used in practice. In practice, the test team must develop the equivalence classes manually. Once the input domain is partitioned, tools exist to generate a quality test suite in a mechanical way.

2.3 Comparison of Strategies for Black Box Testing

In addition to boundary value testing and equivalence class testing, other strategies for developing black box test suites have been proposed. One of these strategies uses decision tables in setting up the different tests. In this paper, this strategy is not discussed in detail. Decision table-bases testing is especially useful when lots of dependencies exist among the different input variables for the software component under test.

Figure 3 compares the number of tests resulting from the different strategies for black box testing. The size of a test suite may be an important criteria, because it determines the time needed to execute the test suite. It may be no surprise that boundary value testing results in test suites involving a lot of elements. The more reasoning is done in setting up test suites, the less tests will be included in the suite. For that reason, the number of test cases resulting from equivalence class testing is quite a lot smaller compared with boundary

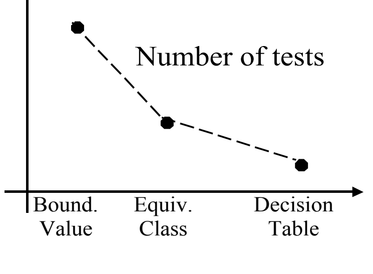

Figure 3: Number of Tests per Black Box Strategy.

value testing. Practice further shows that decision table-based testing results in even smaller test suites.

Figure 4 sketches the effort required to develop test suites by means of the different strategies. It has already been pointed out that boundary value testing is to a large extent a pure mechanical process. The human effort is therefore low. Equivalence class testing cannot be fully automated. The partitioning of the input domain must be done by the test team. Because decision tables are even harder to set up, this strategy requires the biggest effort from the test team.

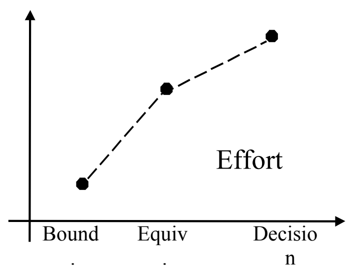

Figure 4: Effort for setting up Test Suites.

The effort required to set up a test suite and the number of tests are important criteria. However, the biggest and most important question is how effective the test suite is in finding errors in the software component under test. This is probably the major drawback of strategies for black box testing. Because the internal of the component under test are not taken into account, this type of question cannot be answered. It is only possible to give some vague guidelines concerning which strategy to use in which case:

- Boundary value testing or equivalence class testing are best used if input variables refer to physical quantities. This is especially true if the input variables are independent of each other. For physical quantities, it is always possible to define an order on the input domain. Moreover, criteria for partitioning such domains are usually easy to find.
- Equivalence class testing and decision table-based testing are best used if input variables refer to logical quantities, and if these variables are independent of each other. Logical quantities are usually difficult to order, which makes it difficult to apply boundary value testing.
- Decision table-based testing is best used if input variables are highly dependent of each other. This is what decision tables have been introduced for in the first place: describing which actions to take under complex conditions, built from complex combinations of more simple conditions.

2.4 Advantages and Disadvantages of Black Box Testing

Most of the advantages of strategies for black box testing are rather straightforward to derive from its principles. The biggest advantage probably is that black box testing is directed towards the outside behavior of software components. After a series of black box tests, there will be some confidence that the tested component indeed behaves as described in its specification.

- The test suite can be designed as soon as the specifications of the software components are completed. Ideally, specifications are built rather early in the process of developing software components. Moreover, because the test suite is independent of the internal of the component under test, there is no need to change the suite each time the implementation of the component changes.
- The test team needs no knowledge of implementation issues, including programming languages or other tools used in the implementation of the component under test. This means that it is possible for project leaders to hire non-software people to work out test suites as soon as specifications have been worked out. The software team itself can then continue working out the implementation of the component.
- Strategies for black box testing help in exposing ambiguities, inconsistencies or gaps in specifications of software components. Proper documentation has always been a problem in the area of software engineering. If a test suite is set up based on that documentation, lots of shortcomings can be revealed, especially if people outside the development team are responsible for testing software components.

The disadvantages of strategies for black box testing are also easily derived from their basic principles. As mentioned above, the biggest shortcoming of black box testing is the lack of objective criteria to evaluate the quality of test suites. As an immediate consequence, there will always remain some doubts concerning the correctness of a software component, for which only some black box tests have been worked out. Other disadvantages of black box testing are briefly described below:

- Without clear and concise specifications of software components, black box test suites are hard, if not impossible, to develop. Because proper documentation is often still lacking, black box testing cannot be applied in some software projects.
- Because black box testing does not build upon knowledge of the internals of software components, it is possible that some paths through the code are left untouched. As a result, software components may turn out to fail, once they are invoked with input values traversing these unexplored program paths. This explains why pure black box testing will never result in 100% confidence concerning the proper functioning of the tested component. White box testing strategies try to resolve this issue.

3. White Box Testing

Contrary to strategies for black box testing, white box testing strategies use the *implementation* of software components to develop test suites. The implementation of a software component describes *how* the component produces output values out of input values. It is worked out in some programming language. If that programming language is an procedural language like C, or an object-oriented programming language like Java or C#, the internals of a software component are said to be *operational*. In building test suites, strategies for white box testing analyze the structure of the source code. White box testing is therefore also referred to as *structural testing*. *Glass box testing* is also often used as a synonym for white box testing.

Because strategies for white box testing start from an in-depth analysis of the source code, they are amenable to rigorous definitions, mathematical analysis and precise measurements. This turns out to be one of the major benefits of white box testing strategies. They will be able to produce some precise figures concerning the quality of a test suite. White box testing is not a new paradigm in the area of software testing. Some strategies were already defined in the mid 1970s, in the context of the paradigm of structured programming.

In this section, we first describe what is meant by an in-depth analysis of source code. In section 3.1, we describe how the source code of a software component can be described in terms of a graph. Such graphs are at the heart of all white box testing strategies. In section 3.2 we give a brief overview of path testing strategies. These strategies introduce coverage criteria that are defined directly on top of the program graph. In section 3.3, data flow testing is introduced as a more in-depth type of white box testing. In this type of strategies, the program path is used as an instrument to derive additional information related to the usage of variables in the analyzed code. This additional information then serves to introduce some solid coverage criteria for test suites. Finally, we discuss the major pros and cons of white box testing in section 3.4.

3.1 Program graphs

White box testing strategies use the source code of software components as the basic information for building test suites. Example 2shows a possible implementation of the method for computing the greatest common divisor, in view of the specification worked out in Example 1. The algorithm starts with throwing an illegal argument exception, if the method is invoked under conditions in which the greatest common divisor cannot be computed. In Java, the throwing of an exception in the body of a method, immediately terminates the execution of the method. After having handled the special cases in which one of the given numbers is `0`, the body of the implementation uses the property that the greatest common divisor of two numbers `a` and `b`, is equal to the greatest common divisor of `a-b` and `b`, provided `a` is greater than `b`. As an example, the greatest common divisor of 63 and 14, is equal to the greatest common divisor of 49 and 14.

```
public static long gcd (long a, long b)
    throws IllegalArgumentException
{
  if ( (a < 0) || (b < 0) || ((a == 0) && (b == 0)) )
    throw new IllegalArgumentException();
  if (a == 0)
    return b;
  if (b == 0)
    return a;
    while (a != b)
      if (a > b)
        a = a - b;
      else
        b = b - a;
    return a;
}
```

Example 2: Implementation of a Java-method for computing the greatest common divisor.

Given the source code of Example 2, white box testing strategies will try to derive a proper test suite from it. Solid criteria for this are discussed in the following sections. One rather intuitive criterion is that each case in the source code should be executed at least once by some element of the test suite. For the algorithm to compute the greatest common divisor, this might lead to the following tests. In section 3.2, we will see that this test suite satisfies the criterion of path coverage.

- `[-3,25]`, `[33,-7]` and `[0,0]` for testing the illegal cases that will lead to the throwing of an illegal argument exception.
- `[0,25]` and `[37,0]` for the testing the correctness of the instructions dealing with the case in which exactly one of the given arguments is `0`.
- `[12,12]`, `[39,13]`, `[17,51]` and `[126,240]` for the actual algorithm for computing the greatest common divisor of non-special values. The first test covers the case in which the loop immediately terminates. The second and third test deal with the case in which one of both input variables is consistently diminished. The last test then deals with the case in which both input variables are diminished during the execution of the loop.

Strategies for white box testing all start from an analysis of the source code under test. For that purpose, the source code is represented in terms of a directed graph. Strategies for white box testing commonly use so-called *decision-to-decision paths* (DD-Paths) [Miller]. Nodes of a such a program graph represent sequences of statements or parts of statements that begin with the *outway* of a decision statement and end with the *inway* of the next decision statement. No internal branches occur in such a sequence. Arrows in the graph then denote possible flows through the source code. Figure 5 shows the program graph for the body of the method for computing the greatest common divisor.

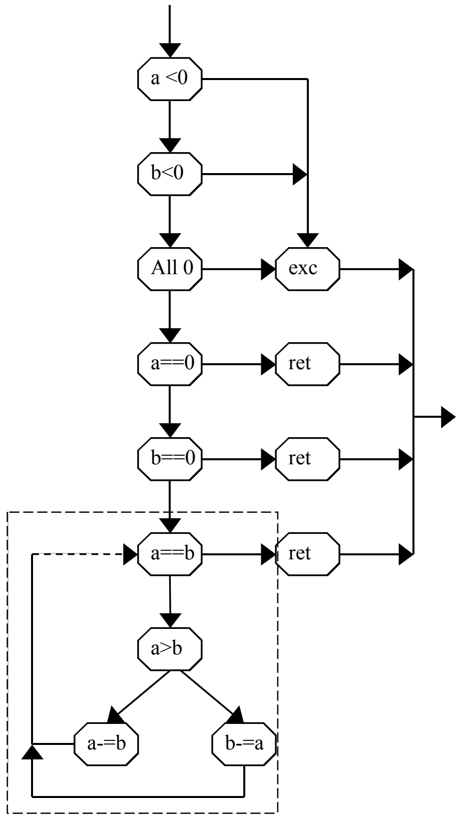

Figure 5: Program graph for greatest common divisor.

- The top node of the graph consists in checking whether the first argument is negative. If it is indeed negative, execution may flow to the node in which the exception is thrown. If the argument is not negative, execution flows to the node in which the second argument is checked for a negative value.
- The bottom of the graph represents the nodes involved in the loop. If the input values are not equal, execution flows to the node searching for the largest value of both. From there on, execution is directed to the node in which the smallest value is subtracted from the largest value. From these nodes, execution flows back to the controlling node of the loop.
- Notice that program graphs are typically constructed in such a way that they all have one entry node and one exit node. This explains why the node in which the exception is thrown and the different return-nodes, proceed with flowing to a common point through which the program graph can be exited.

Loops have always been a bit of a problem in analyzing program graphs. In the context of white box testing, loops are typically represented by a single node, that can be refined into a graph on its own. In Figure 5, this is represented by the dotted rectangle, representing the entire loop as a single node. If that node would node have been worked out in detail, the program graph would flow from the node in which the second input variable is checked to be zero, through the node representing the loop, to the node in which the resulting value is returned.

Program graphs involving DD-paths are a condensed way to capture the structure of the internals of a software component. Instead of producing nodes for each statement or statement-part in the source code, sequences of statements that by definition flow from the one statement into the other are grouped together in a single node. Lots of commercial tools are available producing DD-path graphs for programs written in a great variety of imperative and object-oriented programming languages.

3.2 Path Testing

Once a program graph is available for the software component under test, test suites can be set up using different criteria concerning the flow through the graph. Path testing strategies involve criteria directly related to the program graph itself. All these criteria somehow determine to which extent elements of the program graph are covered by elements of the test suite. Some of the most widely used *coverage criteria* are described below:

1. *Statement coverage*, referred to as C_0, imposes that each *node* in the program graph must have been executed by at least one element in the test suite. In the example graph for the greatest common divisor, this criterion is satisfied by a test suite involving the input values [0 , 0], [0 , 7], [12 , 0], [24 , 16]. These tests are sufficient to visit each node in the program graph at least once.
2. *Predicate Coverage*, referred to as C_1, imposes that each *edge* in the program graph must have been traversed by at least one element in the test suite. This criterion is more strict than statement coverage, because now each branch out of a node must have been followed at least once. In the example graph for the greatest common divisor, this criterion is satisfied by a test suite involving the input values [-3 , 6], [7 , -9], [0 , 0], [0 , 7], [12 , 0], [24 , 16]. Again, it should be easy to see that this suite traverses each edge in the program graph of Figure 5.
3. *Predicate Coverage + Loop Coverage*, referred to as C_2, imposes predicate coverage as described above complemented with some additional criteria concerning possible loops in the source code. The simplest form of loop coverage states that at least one test should jump out of the loop immediately, and at least one test should traverse the loop at least once. A more extended form of loop coverage imposes a test in which the loop is executed a minimum number of times, a test in which the loop is traversed a medium number of times, and a test in which the loop is traversed a maximum number of times. In the example graph for the greatest common divisor, loop coverage would impose at least one additional test using equal values for both input variables (e.g., [12 , 12]).
4. *Path Coverage* starts by examining all possible so-called *basis paths* through the source code represented in terms of a program graph. The notion of a basis path, which is due to McCabe, structures all possible paths through a graph in a vector space. We will not explore these mathematical foundations in this paper. In the example graph for the greatest common divisor, this criterion would be satisfied by a test suite involving the input values [-3 , 6], [7 , -9], [0 , 0], [0 , 7], [12 , 0], [12 , 12], [17 , 51] [39 , 13] and [24 , 16]. Additional tests are added in this case, to cover paths flowing only through the then-part, respectively the else-part of the conditional statement inside the body of the loop.

Nowadays, most quality organizations impose at least predicate coverage in white box testing for software components. Statement coverage is also widely accepted and is mandated by the ANSI Standard 187B.

3.3 Data flow Testing

In data flow testing [Rapps], the program graph for the software component under test is further analyzed. Data flow testing focuses on points at which variables receive values and on points at which values assigned to variables are used. The approach is said to formalize the intuition of testers. Moreover, although data flow testing starts from the program graph, the approach is said to move back in the direction of functional testing strategies.

Strategies for data flow testing start from definitions, reflecting points at which variables are used. In particular, the approach distinguishes between *defining nodes* and *usage nodes* for variables:

- A node in a program graph is a *defining node* for a variable, if and only if the given variable is defined in the statement fragment corresponding to the given node. Defining a variable means assigning a new value to that variable. Input statements, assignments statements, loop control statements and method calls are possible examples of defining nodes. The nodes labeled `args`, respectively `a-=b` are defining nodes for the variable `a` in the program graph of Figure 6. The first node corresponds to the binding of the actual arguments to the formal arguments of the method; the second node assigns a new value to the variable `a`.
- A node in a program graph is a *usage node* for a variable, if and only if the given variable is used in the statement fragment corresponding to the given node. Using a variable means reading the value currently stored in that variable. Output statements, assignments statements, loop control statements and method calls are possible examples of defining nodes. Usage nodes for the variable `a` are marked with a dotted line in Figure 6. The node labeled `a<0` apparently uses the value of the variable `a`; the nodes labeled `ret` are usage nodes because they both return the value of the variable `a` as the ultimate result of the method.

Qualifying nodes as defining nodes and usage nodes for variables may lead to a detection of types of errors referred to as define/reference anomalies. Examples of such errors are attempts to use variables before they are defined, to define variables that are never used and to define a variable several times before they are used. Define/reference anomalies can be detected by a pure static analysis of the source code. Modern compilers signal this type of errors.

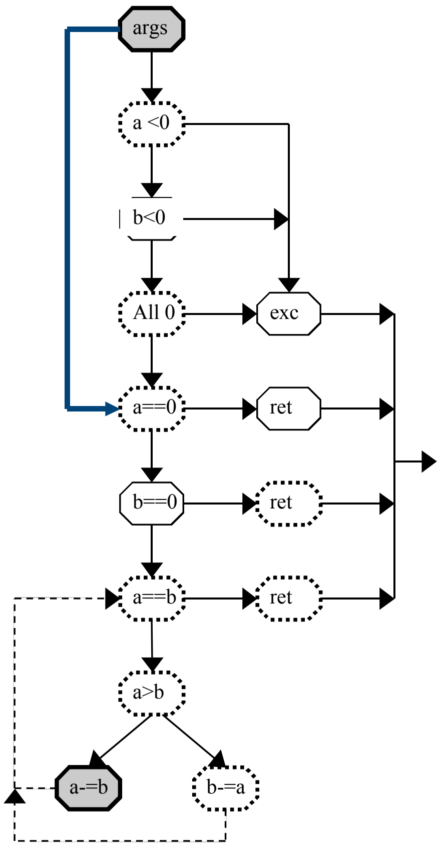

Figure 6: Define/Usage Nodes.

Given the definition of defining nodes and usage nodes for variables, the program graph can be further analyzed, searching for different types of paths:

- A *definition-use path* with respect to some variable is a path that starts with a defining node for the given variable and ends with a usage node for that variable. The path starting at the node labeled `args` in Figure 6, and ending in the node labeled `a==0` is an example of a definition-use path for the variable `a`.
- A *definition-clear path* with respect to some variable is a definition-use path such that no other node than the initial node of the path is a defining node for the given variable. The definition-use path of Figure 6, described above is also a definition clear path for that variable.

Given the analysis of a program graph in terms of defining nodes and usage nodes, and given the set of definition-use paths and definition-clear paths that can be derived from them, a set of coverage metrics can be defined to measure the quality of a test suite. In this paper, we only briefly discuss two such criteria; other exist and new ones will be defined as a result of further research. The message here is that

- A test suit is said to satisfy the *All-Defs criterion*, if and only if for every variable, the set contains definition-clear paths from every defining node of the variable to a use of that variable.
- A test suit is said to satisfy the *All-Uses criterion*, if and only if for every variable, the set contains definition-clear paths from every defining node of the variable to every use of that variable.

In the context of research on data flow testing, other criteria have been defined. We do not discuss all of them in this paper. In the context of this paper it is sufficient to get the feeling that criteria such as the ones above indeed given some appreciation for the quality of a test suite.

At the time of writing, data flow testing has not matured yet. More research is needed searching for better coverage criteria; more experiments must be worked out in order to evaluate how appropriate suggested coverage criteria are. On top of that, further analysis of program graphs may lead to other types of information, that may prove to be very useful in the definition of coverage criteria. One such approach introduces the notion of a program slice as a set of program statements (nodes) that contribute to or affect the value of a variable at a specific node. Whatever coverage criteria are used, there is a general consensus that data flow testing has the potential to replace widely accepted strategies for structural testing on the long run.

3.4 Advantages and disadvantages of White Box Testing

Strategies for white box testing are to a large extent complimentary to strategies for black box testing. As an immediate consequence, advantages of black box testing typically become disadvantages of white box testing, and vice versa. The major advantage of strategies for white box testing is that objective criteria can be defined to quantify the coverage of a test suite.

- Because source code can be transformed into graphs, the mathematical machinery of graph theory can be used to define solid coverage criteria for test suites. Notice that testers themselves do not need an in-depth knowledge of the underlying mathematical theory. They only need to understand simple notions such as defining nodes, definition-clear paths, …
- Because white box testing starts from the source code, testing all parts of a software component is within reach. In particular, it is no longer possible to overlook obscure parts of the source code, because they are an integral part of the internals of the software component under test.

The biggest disadvantage of white box testing is probably that test suites can only be developed late in the life cycle of a software component. Indeed, the implementation of the software component must have been worked out before a test suite can be developed.

- Practice shows that many software projects cannot be delivered on time. Because white box testing can only be started late in the development cycle, there is a potential risk that only a few tests are carried out, resulting in buggy software products delivered to clients.
- Strategies for white box testing also require testers with an in-depth knowledge of implementation techniques. The test team must have an in-depth knowledge of programming languages. In such a context, the test team is often the same team as the team that developed the software component under test. Needless to say, that insiders will often look over the mistakes they made themselves is the products they are testing.

4. Black & White Testing

In surveying strategies for black box testing and white testing, we have seen that both paradigms have their own merits and drawbacks. The major advantage of black box testing is that the functional requirements for a software component are verified without considering its internals. Moreover, black box test suites can be set up rather early in the life cycle of a software component. The major advantage of white box testing is that it is complemented with solid criteria to evaluate the coverage of a test suite.

Black box testing and white box testing are not antipodes. In this paper we propose to combine both approaches in an overall strategy for building adequate test suites for software components. In this way, the advantages of both paradigms are combined, and the disadvantages are eliminated. In out view, the processing of testing a software component should be performed in the following steps:

1. **Step 1**: Develop a test suite based on the specification of the software component under test. This part of the test suite will be referred to as the black test suite. This step can be taken rather early in the development process, as soon as the functional requirements of the component under test have been written down. Notice also that this part of the test suite can be developed by non-software experts. This has the additional advantage that the development team can proceed in parallel with working out the implementation of the software component.
2. **Step 2**: Test the correctness of the software component using the black test suite. These tests can be performed as soon as the implementation of the software component under test has been finished. Typically, a large part of the errors in the implementation of the software component will be detected in this step. It is rather important to correct detected errors, until the software component withstands all the tests in the black test suite.
3. **Step 3**: Extend the test suite for the software component under test using the coverage criterion imposed by your favorite strategy for white box testing. Typically, for complex software components, black test suites will cover about 50% of the actual code. In most cases, it is not important to strive for a 100% coverage. Typically, the process of complementing the initial black test suite with a white test suite will be stopped as soon as a coverage of 80% to 90% is reached.

5. Conclusion

In this paper, we have discussed the basic principles of black box testing and white box testing. We have surveyed some of the strategies supporting both paradigms, and have discussed their pros and cons. Because both paradigms are complementary rather than contradictory, we have proposed a testing process, that starts with the development of a black test suite early in the life cycle, and that is complemented with a white test suite late in the life cycle.

References

- [Eckel] B. Eckel, *Thinking in Java*, Third Edition, ISBN 0-13-100287-2, Prentice Hall, 2003.
- [Frankel] D. Frankel, *Model Driven Architecture*, ISBN 0-471-31920-1, Wiley, 2003.
- [Fowler] M. Fowler, K. Scott, *UML Distilled: A Brief Guide to the Standard Modeling Language*, Second Edition, ISBN 0-201-65783-X, Addison-Wesley, 2000.
- [Meyer] B. Meyer, *Object Oriented Software Construction*, Second Edition, ISBN 0-13-629155-4, Prentice Hall, 1997.
- [Myers] G. Myers, *The Art of Software Testing*, ISBN 0-471-04328-1, Wiley, 1979.
- [Miller] E. Miller, *Tutorial: Program Testing Techniques*, COMPSAC'77 IEEE Computer Society, 1977.
- [Miller] E. Miller, Automated software testing: a technical perspective, *American Programmer*, Vol. 4, No. 4, pp. 38-43, 1991.
- [Mosley] D. Mosley, *The Handbook of MIS Application Software Testing*, ISBN 0-139-07007-9, Prentice Hall, 1993.
- [Rapps] S. Rapps, E. Weyuker, Selecting software test data using data flow information, *IEEE Transactions on Software Engineering*, Vol-SE-11, No. 4, pp. 367-375, 1985.
- J. Warmer, A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, Second Edition, ISBN 0-321-17936-6, Addison-Wesley, 2003.