2011

RECORDING AND EVALUATING INDUSTRY BLACK BOX COVERAGE MEASURES

Tatiana Tokareva

RECORDING AND EVALUATING INDUSTRY BLACK BOX COVERAGE MEASURES

(Spine Title: Recording and Evaluating Industry Black Box Coverage)

(Thesis Format: Monograph)

by

Tatiana Tokareva

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario
December, 2011

THE UNIVERSITY OF WESTERN ONTARIO
SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor	Examiners
James H. Andrews	Ian McLeod
Supervisory Committee	
	Michael Bauer
	Steven Beauchemin

The thesis by
Tatiana Tokareva
entitled

RECORDING AND EVALUATING INDUSTRY BLACK BOX COVERAGE MEASURES

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date	Chair of Thesis Examining Board

Abstract

Software testing is an indispensable part of software development process. The main goal of a test engineer is to choose a subset of test cases which reveal most of the faults in a program. Coverage measure could be used to evaluate how good the selected subset of test cases is. Test case coverage for a program was traditionally calculated from the white box (internal structure) perspective. However, test cases are usually constructed to test particular functionality of a program, therefore having a technique to calculate coverage from the functionality (black box) perspective will be beneficial for a test engineer. In this thesis we discuss a methodology of recording and evaluating the black box coverage for a program. We also implement a black box coverage calculation tool and perform experiments with it using three subject programs. We then collect and analyze experimental data and show the relationship between the two types of coverage and the fault-finding ability of a test suite.

Keywords: Software Testing, Black Box Testing, Equivalence Partitioning, Boundary Value Analysis, Coverage Criteria, Statistical Analysis

Acknowledgments

It would not have been possible to write this thesis without the guidance and support of my supervisor Dr. Jamie Andrews. His ideas, enthusiasm, inspiration and ability to explain things clearly and simply have been invaluable to me.

I would like to acknowledge the financial, academic and technical support of the University of Western Ontario and the Department of Computer Science, and its friendly faculty and staff.

I also would like to thank my husband Alexander for his continued support, patience and understanding, as well as all my family in Russia.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The software development process is a set of activities performed by engineers, managers and testers resulting in the creation of a software product. It usually involves requirements gathering, design, implementation, testing and maintenance activities. Nowadays software systems are becoming more large and complex, with greater risks and costs of a failure. Just imagine, a $2 billion mission to Mars failing because of one software defect. Therefore the importance of thorough software testing, which can help to prevent and eliminate these failures, cannot be underestimated. In this thesis we propose an improvement to the software testing activity. We will discuss the problem of selecting test cases which can detect errors efficiently, and propose a methodology of evaluating the thoroughness of a test suite.

1.1 Software Testing

Software testing is an indispensable part of software development process, which ensures quality and reliability of software under test (SUT), and verifies that SUT

meets specified requirements.

There exist many software testing methodologies, which differ by testing objectives and could produce different results. These methodologies can be distinguished by the level of granularity of software components, by the stage in the software development process during which testing is performed, by testing goals and the qualification of a tester. Unit testing is usually performed by a developer at the coding stage of a project and ensures correct work of individual units of source code, the smallest testable parts of the software system. The goal of integration testing is to ensure that separate modules of an application work correctly as a group, while system testing is performed on a complete integrated system to verify its functionality. Acceptance testing is usually done by the customer after all development and testing has been performed internally, to evaluate the compliance with the requirements of a finished product. Finally, regression testing is done during the maintenance stage, in order to verify that all defects have been fixed, and no new problems have been introduced as part of the maintenance process.

Construction of test cases at any of the described levels is usually based on one of the two fundamental approaches: *white box* and *black box*, which differ by the knowledge that a tester has about the software under test. In the black box methodology testing is based on the requirements and specification, while in the white box methodology testing is based on the knowledge about the code, internal structure, paths and implementation of the software under test. In this thesis we are particularly interested in comparing the effectiveness of these two testing approaches.

1.2 Coverage Criteria

In order to guarantee that a program works correctly, a test engineer needs to execute it with all possible input data combinations and test all logical paths which exist in the program. However even for a system of a small size the number of test cases which cover all input data combinations is infeasible. Therefore, the key issue of testing process is, as defined by Myers [24]: "What subset of all possible test cases has the highest probability of detecting the most errors".

A single execution of a program with the predefined set of environmental conditions and input variables is called a *test case*. The *effectiveness* of a test case is the probability of detecting the errors in a program. In order to evaluate how effective the selected subset of test cases is, the use of *coverage* criteria is essential. Coverage is a measure of what portion of the subject program has been tested, and depending on the testing methodology it could include different coverage elements. Coverage criteria can be used by test engineers in different ways [5]. One way is to have a particular coverage level as a goal during the generation of test cases. Another way is to measure the coverage of the test suite generated manually or by other external mechanisms. In this thesis we concentrate on the second approach.

1.2.1 White Box Coverage

In white box testing (also called *glass box*, *clear box* or *structural* testing), the goal is to create test cases which cover particular lines of code, internal structures, decisions, etc. The most basic white box coverage criterion is statement coverage, in which each executed line of code is considered as a separate coverage element. A statement

is considered to be covered if there exists at least one test case which causes this statement to be executed. Other more sophisticated white box coverage criteria refer to blocks of statements, decisions, conditions within the decision, and paths. Usage of a white box coverage measure is based on the assumption that a test case is more thorough if it causes more elements of a program to be executed.

White box testing has been studied thoroughly in the software testing research field and used extensively by industry practitioners. In the software development industry white box testing is usually applied at the unit testing and integration levels.

One of the advantages of white box coverage is that it is relatively straightforward to measure. For example, measurement of a statement coverage of a program can be done in the following steps. First, at the compilation stage when the source code is translated into the executable object code, special instructions are added to the executable file. These instructions are used to collect the information about executed lines of code in a separate file, when test cases are run against the program. This file could later be used by the coverage calculation tool to produce a coverage report in which each line from the source code is assigned the number of times which it has been executed. There exist various tools to measure statement coverage, block coverage, branch and path coverage for different programming languages; the most popular of them are gcov for C/C++ [17], Cobertura [11] and Jtest [21] for Java.

On the other hand, white box testing has several disadvantages and limitations. First, some defects depend on the environment rather than the code: e.g., running a program in two different browsers might produce a different result: a web page could be displayed correctly in the Firefox browser and be messed up in the Internet Explorer. Second, test case maintenance is required in case of changes in the implementation, because we need to make sure that after changes in the source code the same coverage

level is still achieved by test cases. Finally, it is not possible to cover all executions of loops by test cases. If a test case forces a loop in the source code to be executed three times, we cannot guarantee that the program will produce a correct result on the input which causes a loop to be executed 10 times. Therefore it is important to take into account both measures, black box and white box, as black box looks at the testing from the user's perspective and could reveal faults which could not be found by a test suite with a high white box coverage. However, estimating the black box coverage is less evident, and there do not exist any techniques to estimate it. Therefore, white box testing method alone cannot be used as a guarantee of software quality: it should be supported by functional test cases.

1.2.2 Black Box Coverage

In the black box testing technique (also called *functional testing*) test cases are built solely based on the external information about the program: specification, requirements and design documents. The goal of the black box testing is to verify correctness of the program from the user's perspective. This type of testing is usually applied at higher levels, such as integration, system and acceptance testing, but can also be used as a basis for unit testing.

We have found out that the view on the black box testing methodology in industry and in most research works in this field differs significantly. The majority of the research work which falls into the category of black box techniques concentrates on the generation of test cases from formal specifications or UML diagrams.

In contrast, in industry formal specification of the SUT is available very rarely, therefore major text books written by industry experts place an emphasis on the techniques

for constructing test cases manually or semi-automatically based on the informal specification and requirements. Myers [24], Copeland [12] and other authors consider equivalence partitioning and boundary-value analysis to be the two most basic black box testing techniques, and in this thesis we use these techniques to derive black box coverage elements for the SUT.

1.2.2.1 Equivalence Partitioning

The equivalence partitioning is the most basic black box testing method. According to this method, all possible input values are divided among *equivalence classes.* Each equivalence class involves input variables which are treated in a same way by a program, i.e. if one test case in an equivalence class causes a program to fail, all other test cases in this equivalence class are likely to cause a program to fail, and vice versa. Based on this assumption, a tester could execute a program with only one test case from each equivalence class in order to ensure that a program works correctly. This method allows a great reduction in the number of test cases.

1.2.2.2 Boundary Value Analysis

The equivalence partitioning method is often complimented by the boundary value analysis method, which is the selection of test cases that explore boundary conditions on edges of equivalence classes. It is mostly suitable in case the input is a range of numeric values, either integer or real numbers. This analysis is essential because boundary conditions are places where many of programming errors are made. For example, a programmer could mix up "greater than" with "greater than or equal to" in a conditional expression which will result in an invalid behavior only at the edge

of an equivalence class.

Black box testing also has several disadvantages. First, program specification is not always available, which interferes with the good test case design. Second, a thorough black box test suite can leave some paths in the program unexamined. Hence, white box and black box testing strategies should be used in conjunction.

1.3 Test Case Effectiveness

The ultimate goal of a test engineer is to create test cases which are the most efficient in finding defects in a program. We are interested in comparing the effectiveness of a test case with its black box and white box coverage metrics. *Effectiveness* of a test case is the probability of finding a defect in a program. One of the methods of evaluation of the test case effectiveness is through *mutation.* Mutation is a mechanism of modifying the original source code of a program in small ways. These small mutations usually reflect typical programming errors - wrong operator, value assignment, missing or extra statement. Effectiveness of a test case can be evaluated as the percentage of mutants detected.

It has been shown by Andrews et al. [6] that automatically generated faults can be representative of real faults, therefore the use of mutation in our experiments is considered to be a good way to evaluate the effectiveness of test cases.

1.4 Thesis Focus

Since it is a well-established practice in industry to create test cases which cover particular functionality of SUT rather than particular lines of code, having a technique to evaluate the thoroughness of a test suite from the black box perspective could be advantageous for test engineers. It will allow them to get a high-level view on test suites that they are developing and see if any critical functionality is not covered. Figuring out the relationship between black box and white box coverage measures is critical to software testing research because it will allow software testers to better evaluate a test suite and construct test suites which will be able to find software failures more effectively.

In this thesis we explore a method to evaluate the thoroughness of a test suite from the black box perspective using equivalence partitioning and boundary value analysis techniques - the two most basic test case construction techniques used by industry practitioners. The evaluation method is based on the three main components: Functional Test Specification (FTS), which defines equivalence classes for each input and output variable, as well as multiplicities of components; Log Files, which are produced during the execution of a subject program, and FTS Tool, which matches elements from Log Files with elements from the FTS, and estimates the percentage of elements covered.

We also study the following questions: Does achievement of high black box coverage contribute to the thoroughness of a test suite, and Is it possible to use the black box coverage measure as a predictor of a test case effectiveness? In order to answer these questions we compare black box and white box coverage measures of test cases and randomly generated test suites, and study the relationship between the black box

coverage, white box coverage, test suite size and fault-finding ability of a test suite.

Our experiments have shown that the black box coverage has a statistically significant impact on the effectiveness of a test suite, but it is smaller than the impact of the white box coverage and size of a test suite. We have also found that there exists an exponential relationship between the black box and white box coverage measures, and a test case with low black box coverage is likely to be more effective than the test case with low white box coverage.

1.5 Thesis Organization

Chapter 1 contains an introduction to the topic and relevant background information. We will give an overview of some important concepts as well as related work that has been done studying white box and black box testing approaches in Chapter 2. We will talk about the method of calculating black box coverage, as well as the Functional Test Specification design and the implementation of the black box coverage calculation tool in Chapter 3. In Chapter 4, we will describe subject programs which have been selected for our experiments, as well as design and implementation of the experiments. We will also analyze experimental data, illustrate experimental results and draw conclusions in Chapter 4. In Chapter 5, we will present suggested improvements and future work which could be done in this area.

Chapter 2

Related Work

In this chapter we give an overview of some important concepts of white box and black box testing techniques and test adequacy criteria, and discuss experiments which have been conducted in order to evaluate these techniques.

2.1 White Box Coverage

The terms white box and black box have been used for a long time in industry and were first defined by Myers in his classic book [24]. He also defined and explained the terms statement, decision, condition, decision-condition and multiple condition coverage.

According to Myers, 100% *statement coverage* on code is achieved if for every statement in the code there is at least one test case which executes that statement. A more advanced coverage criterion is the *decision coverage* which looks at the condi-

tional expressions in the `if`, `do`, `while`, etc. statements. 100% decision coverage on code is achieved if for every decision in the code there is at least one test case which causes this decision to be *true* and at least one test case which causes this decision to be *false*. *Condition coverage* is an even more strong criterion, as it considers simple conditions within a decision, which do not contain any logical operators. In order to achieve 100% condition coverage we need to ensure that for every condition in each decision in the code there is at least one test case which causes this condition to be evaluated to *true* and at least one test case which causes this condition to be *false*. Decision-condition and multiple condition coverage types are more advanced extensions of these basic techniques.

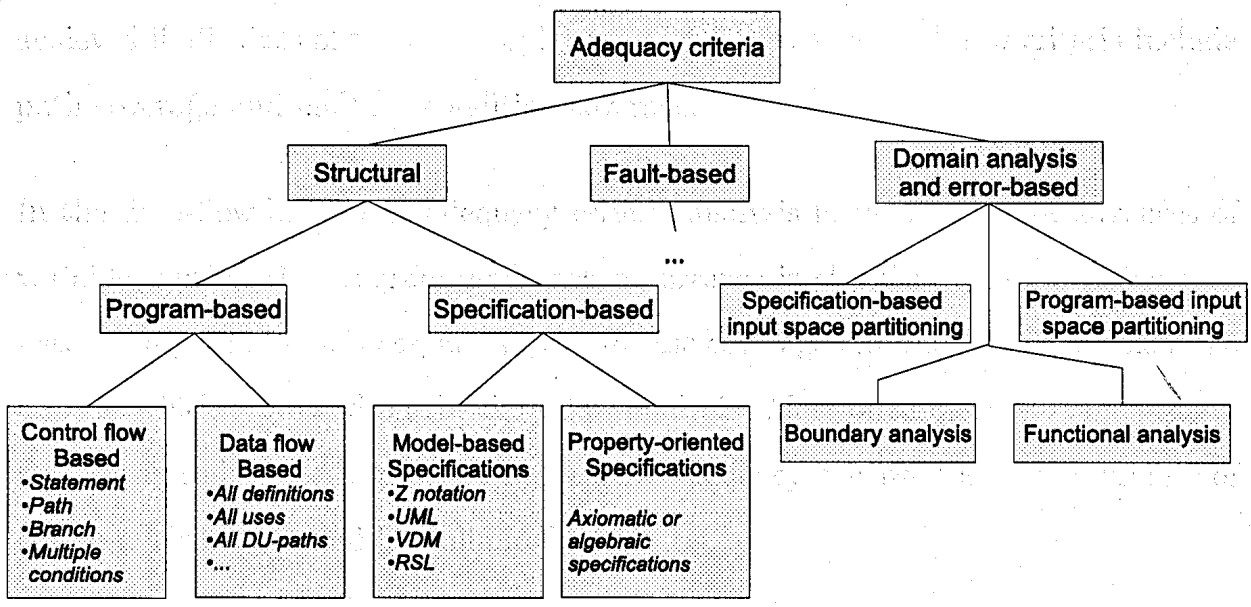
Zhu et al. [31] have created a thorough classification of the existing black box and white box test adequacy criteria based on the research papers in the software testing area. First, the authors define the term *test adequacy criteria* as:

- a stopping rule which determines when enough testing has been performed (e.g. in statement testing, a test set is considered adequate if it causes the execution of every statement in the program);
- a measurement of a test quality (e.g. percent of statements executed), which is similar to the term *coverage criterion*.

The classification of test adequacy criteria is based on the testing approach, and is summarized in Figure 2.1. The following categories are identified:

- *structural testing*, in which coverage elements are based on the structure of the program or the specification;

Figure 2.1 Adequacy criteria structure by Zhu et al.



- *fault-based testing*, in which an adequacy criterion is based on the fault-finding ability of test suites;
- *error-based testing*, which uses domain analysis as a foundation.

The *program-based* and *specification-based* coverage criteria are distinguished within the structural testing category. Program-based criteria correspond to the white box coverage criteria and are divided into the *control-flow* and *data-flow*. Control-flow adequacy criteria are defined based on the *flow graph model* of a program - a graph in which nodes correspond to the linear sequences of statements, edges correspond to control statements or conditions, and each execution of the SUT corresponds to one path in the graph. Based on this notation, the 100% statement coverage criterion can

be achieved if for every node in the flow graph there exists at least one path which covers it. Correspondingly, 100% branch coverage (also called *all-edges coverage*) is achieved if all edges of the flow graph are covered. Other control-flow criteria include path coverage and multiple condition coverage.

In the data-flow-based test adequacy criteria analysis focuses on the occurrences of variables within the program, and each occurrence is classified as a *definition* or a *use*. *All definitions*, *all uses*, and *definition-use* coverage criteria are among the basic criteria which are based on the data-flow analysis. Most of these coverage types are too strong to be used in practice to measure adequacy because the actual number of coverage elements could be unlimited.

In our experiments we use statement coverage, as it is the most basic adequacy criterion, it has been studied thoroughly and there exist a lot of tools to measure it. (Measuring other more strong adequacy criteria can be challenging because of the lack of tool support.) Specifically, we use the `gcov` tool [17] to measure statement coverage of C and C++ programs, and Cobertura [11] for Java subject programs.

2.2 Black Box Coverage

As discussed in the **Introduction** section, the view on the black box testing methodology in industry differs from most research work in this field. In this section we first explain which black box techniques are being developed and studied in the research community, and then focus on the industry perspective on the black box testing.

2.2.1 Black Box Testing Techniques in Research

Test case generation from formal specifications is a well-developed topic in the research on black box software testing techniques. According to Zhu et al. [31], there exist two major approaches to structural specification-based testing: *model-based specifications*, such as Z notations, UML, VDM Specification Language [20] and RSL [18] specifications; and *property-oriented specifications*, such as axiomatic or algebraic specifications.

Generation of test cases based on the formal Z notation specification is one of the well-explored and well-studied techniques. The Z notation language defines components of the system and specifies constraints among them. It was originally proposed by Abrial, Schuman and Meyer [3] and was later used by many researchers to formally define software specification and requirements. Amla and Ammann [4] have developed a method to convert a formal Z specification into a specification in the TSL language [9], from which test cases could be extracted. Stocks and Carrington [28] have used Z specification to build a specification-based testing framework, in which generation of test cases could be automated.

Another well-known approach explores generation of test cases based on the Unified Modeling Language (UML) diagrams. UML is a modeling language and a set of graphic notations to create visual models of object-oriented software systems, developed and maintained by the Object Management Group. There exist many UML diagram types, which could describe both structural and behavioral aspects of a system. Various UML diagrams have been utilized by software testing researchers to generate test cases. Prasanna and Chandran [27] have developed an algorithm for automatic test case generation using UML object diagrams based on a genetic al-

gorithm. Mingsong et al. [23] have proposed a method to automatically generate test cases for UML activity diagrams by comparing execution traces of randomly generated test cases with the UML activity diagrams. Abdurazik and Offutt [2] are using UML collaboration diagrams for static checking and test case generation, which allows for both static and dynamic testing.

There exist a number of research papers which use black box techniques similar to the ones utilized in industry. One of the research papers by Balcer and Ostrand [26] describes a method for generating test cases from a functional specification based on a category-partition method. Within the bounds of this method a tester identifies functional units in SUT, and for each unit defines parameters and its characteristics, as well as objects in the environment which could affect execution of the SUT. Each category is then divided into partitions - different states of a parameter/environment object which could produce different results during execution. This method is similar to the equivalence partitioning black box technique. Information about partitions is written in a certain format called a Test Specification Language (TSL), which is later used by the TSL Tool to produce textual descriptions of test cases.

The second paper by these authors [9] describes improvements to TSL - a more advanced way to define a program's inputs, environment conditions, outputs that it produces, and external changes in the environment. It also introduces an improved version of the TSL Tool which could generate not only a textual description of test cases, but also an executable script for running them and verifying the program's output. At the time of publishing this paper, TSL has been used to test commercial software in the production environment.

The idea of input space partitioning is not unique to the software testing industry, and has different applications in the research papers. For example, Amla and Ammann

[4] have applied category partitioning to Z specifications for test case generation, and the TSL specification language is based on the equivalence partitioning of inputs. However, these techniques are usually used as a supplement for the automated test case generation methods. In this thesis we're interested in exploring the equivalence partitioning and boundary value analysis techniques from the industry perspective and apply these approaches to measure the thoroughness of any test case.

2.2.2 Black Box Testing Techniques in Industry

The majority of software testing text books written by industry experts for test engineers and for students in software testing courses, describe techniques which are used to construct test cases without a formal specification. Myers [24] was one of the first authors to define fundamental black box testing techniques, such as *equivalence partitioning*, *boundary-value analysis*, *cause-effect graphing* and *error guessing*.

He defines a test case design by equivalence partitioning as a two-step process: first, a tester needs to identify the equivalence classes for each of the inputs, and after that define test cases. He gives guidelines for a tester on the construction of equivalence classes, but mentions that it is very subjective, and two testers analyzing the program could come up with different lists of equivalence classes. According to Myers, in order to identify test cases based on equivalence classes, a tester first should cover all valid equivalence classes by test cases, and after that for each invalid equivalence class write a test case in which only one input variable belongs to the invalid equivalence class, and all other variables belong to valid equivalence classes. Usage of only one invalid input variable is essential because if we try to use several invalid values in one test case, an input check on one invalid variable could mask other erroneous-input checks.

Myers [24] defines boundary-value analysis as a selection of test cases which explore situations on and around the edges of equivalence classes. If an input variable specifies a range of numbers, he suggests to write test cases which use both ends of the range as well as values slightly beyond the ends. In addition, he suggests to create test cases which cover boundaries of output variables. Finally, if an input or output variable is an ordered set, attention should be focused on the first and last elements from the set.

Cause-effect graphing is another technique defined by Myers [24], which explores combinations of input variables. In this technique, *cause* is an input variable or a single equivalence class of an input variable, and *effect* is an output variable or a system transformation. First, a graph which links causes and effects is constructed and annotated with system constraints. Second, the graph is converted into a decision table, where each column represents one test case.

The author also points out that the most effective way of testing is by using all strategies together, because each of them targets a particular type of defects. Our approach is based on the equivalence partitioning and boundary value analysis techniques; however, incorporating cause-effect graphing technique might be beneficial and is considered to be one of the future work directions.

2.2.3 Terminology Update

As it was pointed out earlier, equivalence partitioning is one of the most fundamental black box testing techniques, which was created more than thirty years ago. It was originally applied to small utility programs with text-only Unix-like command line interfaces, where main sources of input were command line parameters and text files.

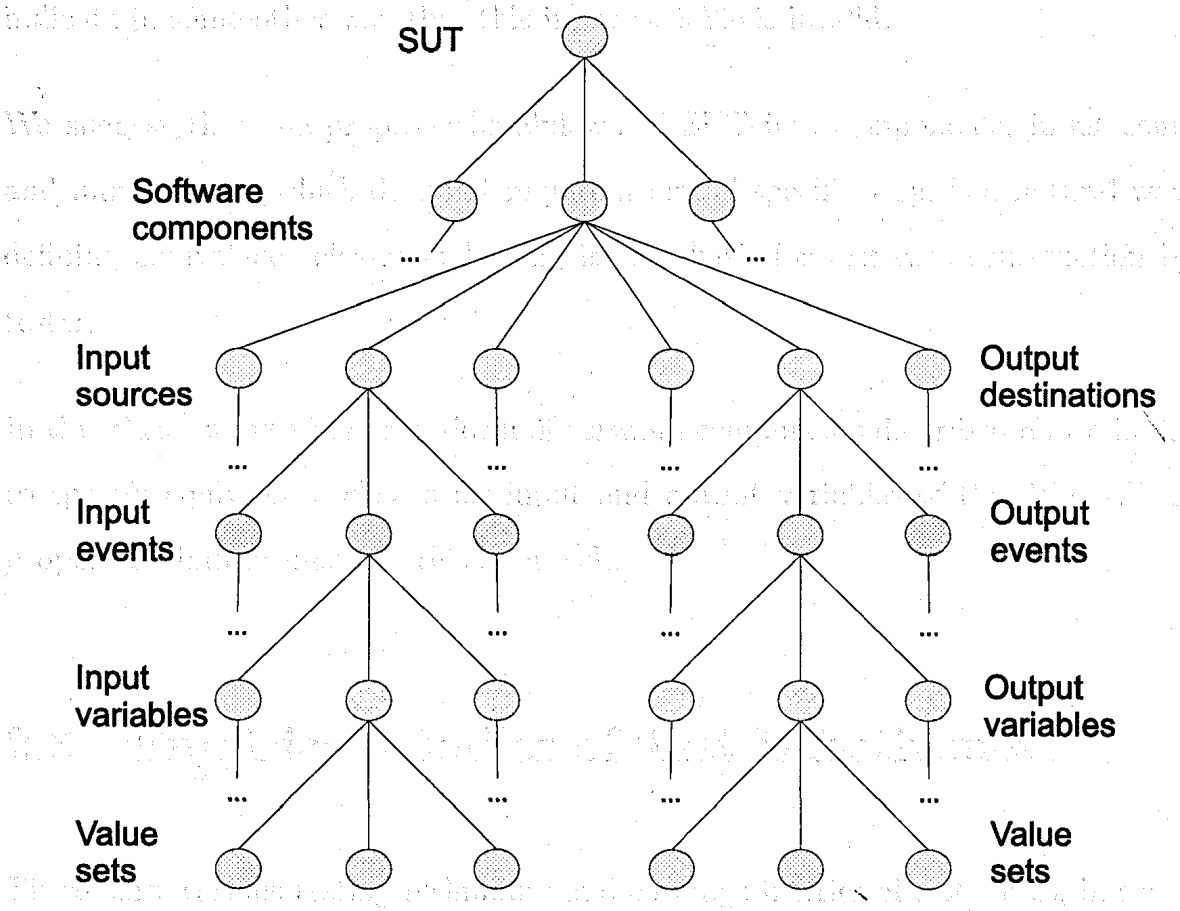
Both Myers [24] and Balcer et al. [9] use relatively small programs with no more than ten parameters as examples, and there was no need to develop a more advanced classification of input variable types.

Nowadays software systems have become much more advanced; they could include separate modules and components, each of which could consist of multiple GUI forms, and use numerous sources of inputs such as files, databases, network connections, etc. However, modern software testing text books [12] are still using the same terminology and apply it to small sample programs.

Therefore there was a need to refine the equivalence partitioning approach and update terminology. Andrews [7] has proposed an equivalence partitioning scheme which takes into account the complexity of software systems and for each input variable defines in which software component it appears, what source of input was used and what input event has caused this input variable to be processed by the SUT. The same approach is applied to the output variables. The graphical representation of the revised equivalence partitioning on inputs and outputs is presented in Figure 2.2.

A *software component* is an individual module of a single system, a software package or a web service which provides a set of related functions. For example, in a client-server system a client and a server could be considered as two separate software components. A *source of input* is anything external to the SUT, provided by the user and which could influence the behavior of the SUT. Sample sources of input are command line parameters, standard input, files, and the graphical user interface (GUI). Correspondingly, an *output destination* is something created or modified as a result of the execution of the SUT, such as standard output, error logs, the GUI, and output files. An *input event* is any event which involves any of the SUT's input sources. It could be a menu selection, button press, command typed by the user,

Figure 2.2 Structure of coverage elements by Andrews et al.



program launch, etc. Accordingly, an *output event* is an event of producing output by the SUT, which involves one output destination. Examples of output events are messages presented to the user, data written to a file or a database, or messages sent.

Input and *output variables* are the most basic input and output elements in the equivalence partitioning method; they are usually strings, numbers or boolean values. Examples of input variables are user name, port number, side length, month number, day number, and column width. Each input and output variable could be broken down to *value sets*, which can be considered as value-level equivalence classes. Value sets of input variables, in contrast with output variables, could be invalid, which means

that when a system receives such input, it is supposed to give an error message or indicate in some other way that this input variable is invalid.

We suggest that the proposed breakdown of SUT into components, input sources and input events, which does not require a formal specification, is a natural way of defining equivalence classes and using it as a basis for test case construction by a tester.

In this thesis we use the breakdown of software components described above in order to specify equivalence classes for input and output variables of the SUT. We also propose an improvement to this approach.

2.3 Empirical Studies of Test Effectiveness

There exist various testing techniques and coverage metrics of test suites, but as the ultimate goal of a testing process is to find faults in a program, the main concern of a test engineer is "how achieving high coverage contributes to the test case effectiveness". Multiple studies have been performed which support the correlation between various white box coverage criteria and test suite effectiveness.

Frankl and Weiss [15] have performed an experiment in which they have compared the effectiveness of the dataflow-based all-uses and controlflow-based all-edges test adequacy criteria for small Pascal programs with existing faults. They have measured the percentage of executable edges and definition-use pairs for each test suite, and counted how many program faults were revealed by this test suite. The results of the experiments have shown that the fault-finding ability of a test suite is positively correlated with both all-uses and all-edges adequacy criteria only for half of the

subject programs.

A similar experiment was performed by Hutchins et al. [19] in which they have used moderate-size C programs with seeded faults, and a different experimental setup. They have found out that test sets which achieve coverage levels of 90% or more have much higher effectiveness than randomly chosen test sets of the same size. Frankl and Iakounenko [16] have studied the relationship between the effectiveness of randomly generated test suites, dataflow-based definition-use and controlflow-based decision test adequacy criteria, and have observed a similar pattern. Faulty versions of a real-world C program for antenna configuration were used in this experiment.

In a more recent study Andrews et al. [25] have studied the relationship between effectiveness, white box statement coverage and the size of a test suite. They have prepared a much larger set of faulty versions of subject programs generated automatically through mutation, which allowed them to prove statistical significance of results and also made experiments reproducible. The experiments indicate that both size and coverage influence test suite effectiveness; however, the relationship between these three variables is not linear. Instead, a linear relationship among variables log(size), coverage and effectiveness was observed for all subject programs. Within the bounds of this thesis we perform experiments which build on this work, and determine if adding the black box coverage to the model could make it more accurate, and if a nonlinear relationship among the black box coverage, size and effectiveness of a test suite still holds.

Another goal of this thesis is to compare the white box and black box coverage of test suites and individual test cases. While there do not exist any studies directly comparing the black box coverage with white box coverage of a test suite, a recent publication by Yu et al. [30] studies the white-box coverage of a test suite which was

generated from the functional black box specification, and therefore achieves 100% black box coverage. Path coverage was selected as a white box coverage measure, as it is the strongest criterion. The study has revealed that a specification-based test suite may not take into account all implementation details, so the comparison was made using only "spec-related paths". The study showed that a spec-based test suite covers about 97% of spec-related paths in the code, which is a very high number.

2.3.1 Mutation

Experimental results of running test cases on the SUT are usually used as an empirical assessment of a test case effectiveness. However, there are several problems connected to the design of experiments. First, a researcher needs to have a correct version of a program as well as several faulty versions, where each version contains only one fault. Finding and preparing such faulty versions is very difficult and time-consuming. Second, the number of faulty versions might not be enough in order to achieve statistical significance in the experiment. Therefore, many researchers are creating faulty versions of subject programs by introducing faults either automatically or by hand. Preparing the necessary number of faulty versions with hand-seeded faults could also take a long time, so it is more efficient to automate this process. In order to produce automatically-generated faulty versions, the original source code of the program is automatically modified in small ways to produce a program mutant. These small modifications are called mutation operators, and reflect typical programming errors: wrong operator in the logical condition, incorrect value assignment, missing statement, etc.

DeMillo et al. [13] have originally proposed an idea of using mutants to measure test

case adequacy, and have implemented a prototype mutation system for the FORTRAN language. This idea was later explored by DeMillo in collaboration with Offutt [14]. Andrews et al. [6] have performed experiments in order to identify if mutation is an appropriate tool for the empirical evaluation of testing techniques. They have compared the ability of test suites to detect real, hand-seeded and automatically generated faults, and have found out that mutants can provide a good indication of the fault detection ability of a test suite, when using carefully selected mutation operators and after removing equivalent mutants [6]. Therefore mutants are representative of the real-world faults, and can be used to assess the effectiveness of test cases.

Chapter 3

Black Box Coverage Calculation Method

In this chapter, we go into detail about the design and implementation of the black box coverage calculation approach. We describe three main components of this approach: *Functional Test Specification* or *FTS*, log files, and *FTS Coverage Calculation Tool* or *FTS Tool*. We also write about the architecture of the FTS Tool, describe some important classes and methods, and give details of the algorithms for calculating the coverage of a test suite based on the log files and FTS.

3.1 Overview

As mentioned in the Related Work section, black box testing techniques, which are widely used in the software development industry, are not studied thoroughly in the software testing research field. Moreover, there does not exist a tool to measure the

coverage of a test suite from the black box perspective. Therefore the main motivation of this thesis is to develop an approach to measure industry black box coverage based on the industry equivalence partitioning and boundary value analysis methods.

The first part of this approach is the construction of the Functional Test Specification for the SUT, which defines equivalence classes of input and output variables, as well as higher-level elements of the software system to which these variables belong. In order to determine which elements from the FTS have been used in a particular test case, we require that the SUT produces log files in a particular format during execution. Log files will contain information about values of input and output variables which have been used in a particular test case, as well as events, sources of input and components of the SUT, in which these variables appeared. This information will allow us to calculate the ratio of the number of tested elements to the total number of elements for each element type. As we are also interested in determining the number of repetitions of each element from the FTS, we're going to organize logs in such a way that we will be able to determine how many times a particular element appeared within the parent element. Finally, the FTS Tool will perform matching of the FTS with the log files and calculate the following coverage types:

- *Simple existence coverage*, according to which an element is considered to be covered if it appears in the log file at least once.
- *Multiplicity coverage*, which takes into account the number of repetitions of each element from the FTS.
- *Boundary value coverage*, which is calculated for each equivalence class consisting of the range of numeric values, and checks if boundary values appear in the log file.

A detailed description of how the coverage measures are collected is given in Section 3.4.3.

3.2 Functional Test Specification

Functional Test Specification is a way to capture the result of applying the equivalence partitioning method to input and output variables of the software system. As mentioned in the Related Work section (see Section 2.2.3), modern software systems are very complex and can consist of many components with multiple GUI forms and sources of input. Therefore it is not enough to specify only input and output variables of the system. Instead, we're going to use the breakdown of the software system into elements proposed by Andrews [7].

In our coverage calculation approach we would like to consider not only the appearance of particular system elements, but also the number of repetitions of elements of a particular type. For example, if a utility requires only one input file, the tester will likely create at least one test case with one file given as input, one test case with no files given, and possibly one test case with two or more files given. One possibility of tracking the number of repetitions would be to create an additional artificial input variable which would represent the number of repetitions of a particular element. However, another more consistent approach is to specify a *Multiplicity* property for each of the elements for which it is necessary. The multiplicity property could be applied to any of the coverage elements except value sets, and could be broken down into valid and invalid value sets, just as any other input variable. In the previous example, the valid multiplicity of the "input file" element will be "1", and two invalid multiplicities will be "0" and "2". If a tester is specifying a multiplicity property

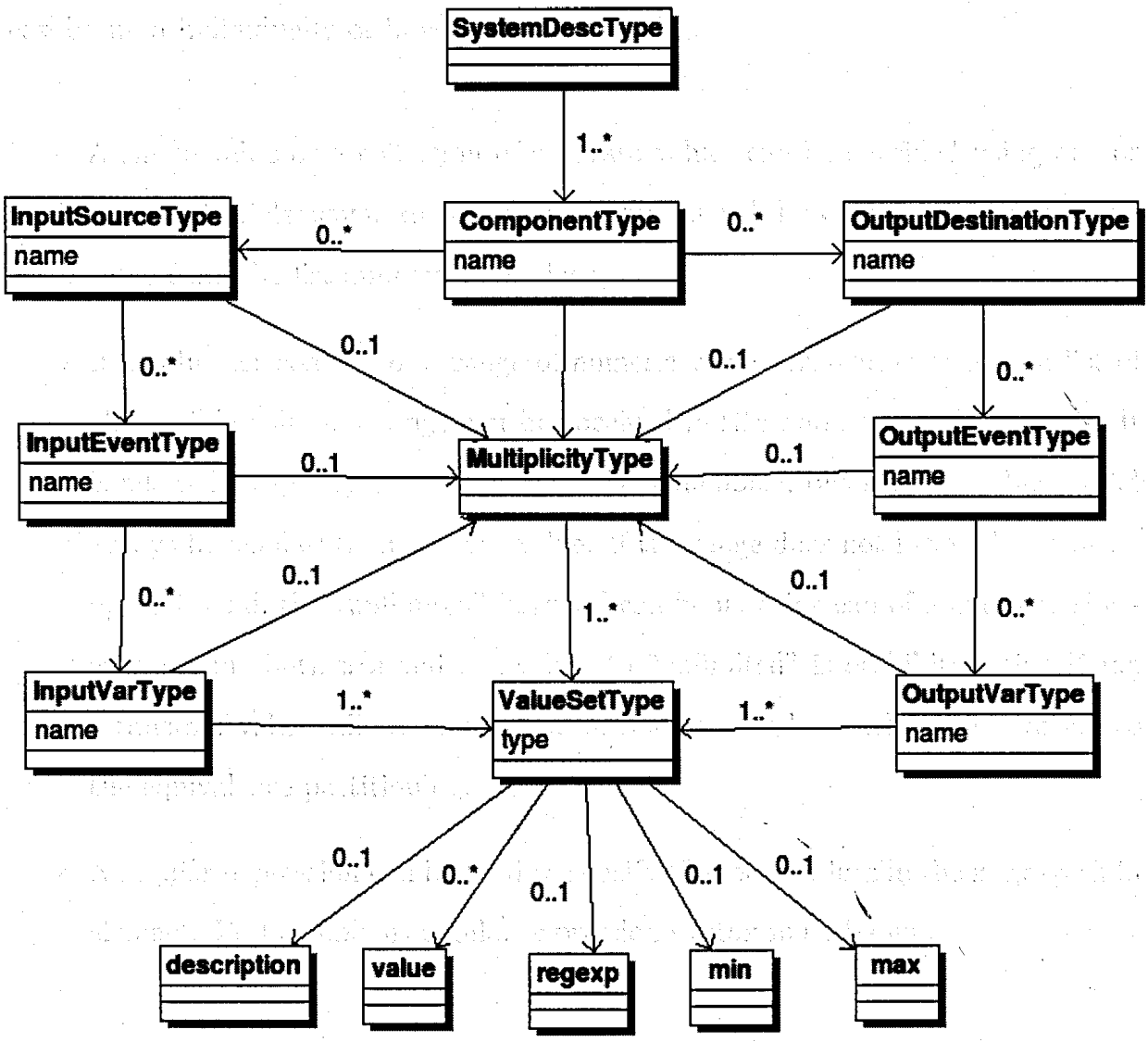
for a particular coverage element, this implies that he believes that in order for this element to be tested thoroughly, for each multiplicity value set there should be at least one test case with the corresponding number of elements of this type.

After a tester has analyzed a program's structure, its input and output variables, and possible breakdown into value sets, he should store this information in some convenient form. One of the most popular formats for storing information in a structured form is the Extensible Markup Language (XML) [10]. We therefore chose XML for the representation of the FTS. It is a textual data format which allows users to represent structured information using their own custom markup scheme. We wrote an XML schema which corresponds to the FTS. An XML schema is a set of restrictions which are assigned to a particular XML file, and can be used to verify the validity of XML. A visual representation of this schema is shown in UML format in Figure 3.1.

`SoftwareDesc` is the root element, which can contain one or more `ComponentType` elements. Each `ComponentType` element can include zero or one `Multiplicity` elements, and zero or more `InputSourceType` and `OutputDestinationType` elements. A `Multiplicity` element specifies the number of repetitions of the parent component; it consists of one or more `ValueSetType` elements and can be a child element of any other element except itself and `ValueSetType`. Each `InputSourceType` element can contain zero or more `InputEventType` elements, which in turn can contain zero or more `InputVarType` elements. Each of the `ComponentType`, `InputSourceType`, `InputEventType` and `InputVarType` elements must have a "name" attribute to specify a unique component name, which is used while matching specification with logs.

`InputVarType` must contain at least one `ValueSetType` element, which represents an equivalence class for this variable. `ValueSetType` does not have to have a unique name and can be uniquely identified by its set of values. An optional `description`

Figure 3.1 XSD schema



child element can be used to write a comment or description. `ValueSetType` also has an optional `type` attribute, which can be assigned one of two values: "valid" or "invalid". If this attribute is not specified, it is assumed by default that the value set is valid. Valid value set contains input values which are expected by a program as valid inputs and make the program operate in a normal mode. In contrast, invalid value set contains values which will cause error handling in the program or will make the program indicate to the user that such input will not be handled correctly.

We have designed three options to specify the contents of a particular value set, which can be used individually or in conjunction:

- A single value or a collection of separate values can be specified using one or more `value` elements. In this case a value set will be considered tested if any value from the list appears in the logs.
- If a value set consists of a range of numeric values, instead of writing a list of all possible values, a range can be specified in `min` and `max` child elements. It is allowed to specify both integer and float numbers, but the `min` value should always be smaller than the `max` value. If the range does not have a lower or an upper bound, the "unlimited" keyword can be used instead of a number. However, setting both `min` and `max` values to "unlimited" is prohibited. Specifying a range of values allows us to perform boundary value analysis in addition to the equivalence partitioning.
- A regular expression can be used to specify the set of values in the `regexp` child element. Perl-compliant regular expression syntax must be used.

Correspondingly, `OutputDestinationType` contains at most one `Multiplicity` element and zero or more `OutputEventType` elements. `OutputEventType` consists of at most one `Multiplicity` element and of one or more `OutputVarType` elements. `OutputVarType` consists of zero or one `Multiplicity` elements as well as one or more `ValueSetType` elements, which are similar to those used in the `InputVarType`, but can only contain valid value sets.

Figure 3.2 shows an excerpt from the FTS specification for the mastermind game server, one of the subject programs which will be described in detail in Section 4.2.3.

Figure 3.2 Sample Functional Test Specification

```
<Component name="Server">
  <InputSource name="CommandLine">
    <InputEvent name="ServerInitialized">
      <InputVar name="PortNumber">
        <Multiplicity>
          <ValueSet type="invalid">
            <value>0</value>
          </ValueSet>
          <ValueSet type="valid">
            <value>1</value>
          </ValueSet>
        </Multiplicity>
        <ValueSet type="invalid">
          <min>0</min>
          <max>1023</max>
        </ValueSet>
        <ValueSet type="valid">
          <min>1024</min>
          <max>65535</max>
        </ValueSet>
      </InputVar>
    </InputEvent>
    <InputEvent name="GameInitialized"/>
  </InputSource>
</Component>
```

According to the specification, exactly one port number value should be specified in the command line in order to initialize the server. Port numbers from 0 to 1023 are considered to be invalid, and numbers from 1024 to 65535 - valid.

3.3 Log File Format

After constructing the program's FTS specification, we need to determine which of the specified coverage elements have been tested during the execution of the SUT, i.e.

Figure 3.3 Format of log files

```
Component <ComponentType> <componentName>
InputSource <componentName> <InputSourceType> <inputSourceName>
InputEvent <componentName> <inputSourceName> <InputEventType>
  <InputVarName1>=<inputVarValue1> ... <InputVarNameN>=<inputVarValueN>
OutputDestination <componentName> <OutputDestinationType>
  <outputDestinationName>
OutputEvent <componentName> <outputDestinationName> <OutputEventType>
  <OutputVarName1>=<outputVarValue1> ... <OutputVarNameN>=<outputVarValueN>
```

which components were used, what events occurred during execution, what variable values were set, and what output was produced by the program. Depending on the program type, structure and functionality, we could extract this information from the execution logs, standard output, database transactions, or GUI components. As we would like our approach to be applicable to a wide range of programs implemented in any programming language, we can not use any existing standard logging mechanism in order to collect this information automatically. Therefore instrumentation of the SUT, which will produce log files in the appropriate format, is required as part of our approach.

Instrumentation could be done either by a tester or a developer, as it involves simple operations of writing necessary information into the file, and does not require special knowledge either about the system's internal structure, or about the programming language used. A separate file with the unique name will be created during each execution of the program, so that one test case will correspond to one log file. A log file will consist of separate lines; each of these lines will contain information about a particular coverage element, and will be written to the file when the corresponding event happens during the program's execution. The format of log lines is presented in Figure 3.3.

We have defined five different types of log lines, and each of them starts with a keyword which identifies the type of the coverage element from the specification. The first line starts with the `Component` keyword, followed by the name of the `ComponentType` from the specification and a unique component name, which will be used as a reference when defining other elements. The second line starts with the `InputSource` keyword, followed by a unique name of the component to which it belongs, the name of the `InputSourceType` from the specification, and a unique name. The third line starts with the `InputEvent` keyword and defines an input event for a particular component and input source. It does not have a unique name because it is not referenced further in other types of log lines. It also defines tuples of input variable names which have been used in this input event together with their values, where zero, one or more input variables of the same type could be specified in the same line. Output destinations and output events are defined similarly to input sources and input events.

Some string values of input variables can contain whitespaces, e.g. an input variable for a user name "John Smith". In our logging format a whitespace is used as a separator, and a tester needs to take this into account while constructing logs: before writing a value into a log file he should check if it contains whitespaces, and put it into double quotes if necessary.

Figure 3.4 shows a sample of three log files for a mastermind game server which correspond to three test cases:

- A server was launched with a valid port number 65535, and a new game was initialized.
- A port number has not been provided.
- An invalid port number 80 was used.

Figure 3.4 Sample log file

```
testcase1.txt:
Component Server server
InputSource server CommandLine cl
InputEvent server cl ServerInitialized PortNumber=65535
InputEvent server cl GameInitialized

testcase2.txt:
Component Server server
InputSource server CommandLine cl
InputEvent server cl ServerInitialized

testcase3.txt:
Component Server server
InputSource server CommandLine cl
InputEvent server cl ServerInitialized PortNumber=80
```

3.4 FTS Coverage Tool

3.4.1 Overview

We have developed a Java utility program which matches coverage elements from the FTS with the coverage elements which appear in the log files, and produces a coverage report. It is called *FTS Coverage Tool* and was developed in the Java Development Kit (JDK) v1.6.17. It is compatible with all versions of JDK 1.6 and can run on any operating system with the Java Runtime Environment (JRE) installed.

Compiled Java class files are packaged into an archive `fts.jar` which can be executed by the Java application launcher. The required parameter for the FTS Tool is the path to the FTS specification file, which should be passed after a keyword `-xmlspec`. The second required parameter is the path to the location of log files, which could be specified in two ways. A tester could provide a list of log file names separated by

a whitespace and preceded by the keyword `-filelist`. Another option is to specify a directory name in which log files are stored, by using the keyword `dir`. Options `-filelist` and `-dir` cannot be used together. By default, FTS Tool calculates coverage for each type of coverage element and prints it in human-readable form. Specifying an optional parameter `-listce` makes the tool printing the list of coverage elements, one coverage element per line, with the indication if this element was covered or not. For example, if we launch the program with the following parameters:

```
java -jar fts.jar -xmlspec ../mastermind/mastermind.xml
    -filelist ../mastermind/logs/testcase1.txt
```

FTS Tool will produce the default coverage report for each type of coverage element for only one test case `testcase1.txt` using the `mastermind.xml` specification file.

3.4.2 Output Format

The coverage report produced by the FTS Tool can have two different formats. The sample of the report file with the default output is presented in Figure 3.5.

The report is broken down into four parts. The first part contains coverage values for three main types of coverage: simple existence, multiplicity and boundary value coverage. The ratio of the number of covered elements to all elements of this type as well as a percentage value are given. A breakdown into valid and invalid coverage elements is performed for the multiplicity coverage. The second section presents a detailed report on each type of simple existence coverage elements: components, input sources, input events, etc. The third section contains a list of element IDs which have not been covered during testing. This list includes all types of coverage elements -

Figure 3.5 Sample FTS Tool output

```
----- Total coverage -----
Simple existence coverage: 32/79 = 40.51%
Total multiplicity coverage:   6/12 = 50.00%
Valid multiplicity coverage:   6/11 = 54.55%
Invalid multiplicity coverage: 0/1 = 0.00%
Boundary value coverage: 0/12 = 0.00%

----- Detailed simple existence coverage -----
Components existence coverage: 1/1 = 100.00%
Input sources existence coverage: 2/2 = 100.00%
Input events existence coverage:  4/7 = 57.14%
Input vars existence coverage:  8/11 = 72.73%
Input value sets existence coverage:  11/49 = 22.45%
Valid value sets existence coverage:  11/28 = 39.29%
Invalid value sets existence coverage:  0/21 = 0.00%
Output destinations existence coverage: 2/2 = 100.00%
Output events existence coverage: 2/4 = 50.00%
Output variables existence coverage: 1/1 = 100.00%
Output value sets existence coverage: 1/2 = 50.00%

----- Missing coverage elements -----
OutputEvent Concordance.StdOut.OutOfMemoryMsgPrinted
Valid ValueSet Concordance.InputFile.FileLoaded.Property: 'Empty file'
InputEventMultiplicity Concordance.CommandLine.HelpOption multiplicity '1'
Max Boundary OutputValueSet Concordance.FileSystem.OutputFileCreated
    .WordsCount: 'Integers'
...
----- Not matched variables -----
InputVar InputFileName=invalid.txt
```

simple existence, boundary value and multiplicity, and in order to distinguish different elements, a unique identifier (ID) is constructed for each element. The ID includes not only the element name, but also names of its ancestor elements. For example, a unique ID for a ValueSetType element is constructed in the following way:

```
<ComponentType>.<InputSourceType>.<InputEventType>.<InputVarType>:
   [<description>|<value>|<regular expression>|from <min> to <max>]
```

As ValueSetType does not have a unique name assigned to it, its description from the FTS will be displayed. If the description was not specified, either a value, a range of values or a regular expression will be displayed, depending on the method which was used to construct this ValueSet in the FTS. In case of multiplicity value sets, we need to specify both the unique ID of the element to which this multiplicity property belongs, and an identifier of a multiplicity value set. For example, the multiplicity value set for an input variable can be specified in the following way:

```
InputVarMultiplicity <ComponentType>.<InputSourceType>.<InputEventType>
    .<InputVarType> multiplicity <ValueSetUniqueID>
```

When printing out information about the boundaries, we need to specify a type of boundary (minimum or maximum), value set type and a unique ID of this value set:

```
[Min|Max] Boundary [InputValueSet|OutputValueSet|MultiplicityValueSet]
          <ValueSetUniqueID>
```

The final section of the report displays coverage elements from the log file which did not match any value sets from the specification. This section helps to troubleshoot any problems, such as an error in the logging or in the specification. For example, if a particular value appears in the list of not matched values, but instead should belong to some value set, a tester might have to review the specification, and make a modification to the description of this value set.

Figure 3.6 Calculation of simple existence components coverage

```
for each ComponentType c from spec do
    for each TestCase tc from logs do
        N <- number of components of type c in tc
        if (N > 0) then c.tested <- true
```

3.4.3 FTS Coverage Calculation

The main functionality of the FTS Tool is the calculation of a test suite's coverage, which is based on the comparison of the specification with the log files. FTS Tool's functionality includes calculation of three types of coverage: simple existence, multiplicity and boundary coverage.

3.4.3.1 Simple Existence Coverage

Simple existence coverage is calculated for each type of coverage elements defined in the XML specification (see Section 3.2), and the general rule is to consider a coverage element to be tested if it appears in the log file at least once. For example, in order to calculate coverage of software components, for each component type defined in the specification, we need to execute the following: for each test case (or each log file) we count the number of components of this type which appear in a particular test case, and mark this component type as "tested" if the number is greater than zero. After examining each component type from the specification, we can count the number of tested components, divide it by the total number of components and present to the user in a specified format. Pseudo code of the simple existence components coverage calculation algorithm is shown in Figure 3.6.

The rule described above is applied to most of the coverage elements; however, the coverage of input variables and value sets has additional constraints, and is calculated in a different way. Specifically, we consider an input variable from the log file to contribute to the coverage only in the following cases:

- All values in the corresponding input event from the log file are from valid value sets, and multiplicity of all input variables in this input event is valid.
- One value in the corresponding input event from the log file is from an invalid value set, and multiplicity of all input variables in this input event is valid.
- All values in this input event are from valid value sets, and multiplicity of one input variable is invalid.

A good test case design assumes that each invalid condition is tested in a separate test case: if we run a program with several invalid conditions at once, we will not be able to determine which invalid condition has caused error-handling in the program. Therefore, we consider input variables to be tested if they appear in an input event which corresponds to a valid test case, where either all variables have valid values and multiplicities or only one variable has invalid value or multiplicity. For example, if a program takes as input a day number (1-31) and a month number (1-12), then providing an invalid day and a valid month will be considered as a valid test case, and will be counted towards the total coverage. An input with two valid day values and one valid month value will have an invalid multiplicity of a day variable, and will also be counted towards the total coverage. However, providing an invalid day and invalid month values will not be considered as a valid test case.

As shown in Figure 3.7, in order to check for these constraints, while iterating through all input events of a particular type, we execute the following steps:

Figure 3.7 Calculation of simple existence input variables coverage

```
For each ComponentType ct from spec do
  For each InputSourceType ist from ct do
    For each InputEventType iet from ist do
      For each InputVarType ivt from iet do
        For each InputEvent ie of type iet from logs:
          n1 <- number of invalid input variables in ie
          n2 <- number of invalid multiplicities of input variables in ie
          If (n1 + n2 < 2)
            N <- number of input variables of type ivt in input event ie
            If (N > 0) then ivt.tested <- true
```

1. Calculate n1 - the number of input variables which have invalid values in this input event.

2. Calculate n2 - the number of input variables which have invalid multiplicity. In order to check if any input variables in this input event have invalid multiplicity, we need to iterate through all input variable types, which are defined in the specification for an input event of this type, count the number of input variables, and compare it with the multiplicity value sets, if there are any. If the number matches an invalid multiplicity value set, we consider this input variable to have an invalid multiplicity.

3. If n1 + n2 is less than two (it covers situations when both values are zero or only one of the values is one), we proceed further to calculate simple existence coverage of the corresponding input variable type.

We consider input variables to be covered if they appear in a valid test case. However, in order to check for particular values of input variables, we need to apply a more strict rule. If a test case contains an invalid value or invalid multiplicity, it causes the

Figure 3.8 Calculation of simple existence value sets coverage

```
For each ComponentType ct from spec do
  For each InputSourceType ist from ct do
    For each InputEventType iet from st do
      For each InputVarType ivt from iet do
        For each valid ValueSetType vvst from ivt do
          For each InputEvent ie of type iet from logs:
            n1 <- number of invalid input variables in ie
            n2 <- number of invalid multiplicities of input variables in ie
            If (n1 == 0 and n2 == 0)
              For each InputVar iv from ie
                If (vvst.match(iv.value) = true) then
                  vvst.tested <- true

        For each invalid ValueSetType ivst from ivt:
          For each InputEvent ie of type iet from logs:
            n1 <- number of invalid input variables in ie
            n2 <- number of invalid multiplicities of input variables in ie
            If (n1 == 1 and n2 == 0)
              For each InputVar iv from ie
                If (ivst.match(iv.value)) then
                  ivst.tested <- true
```

program to execute error-handling code for the invalid value, while the functionality which involves other valid values is not executed. Therefore, for a test case which contains an invalid value or multiplicity, we can only say that the value set which corresponds to the invalid value was tested. In order to cover other valid value sets, we need to execute a test case where all values are valid. Therefore, we apply the following coverage calculation rule:

- A valid value set is considered to be covered if all values in the corresponding input event are from valid value sets and multiplicity of all input variables is valid.

- An invalid value set is considered to be covered if all other values in the corresponding input event except the current one are from valid value sets and multiplicity of all input variables is valid.

In order to check for these constraints, we calculate the number of invalid input variables and the number of invalid multiplicities of input variables, similarly to the input variables coverage calculation. As shown in Figure 3.8, we process valid and invalid value set types separately. For valid value set types, we check that there are no invalid variables and invalid multiplicities in the current input event; for invalid value set types, we check that there is exactly one invalid value set and no invalid multiplicities.

3.4.3.2 Multiplicity Coverage

Multiplicity is an optional property in the FTS specification, and is usually specified only for a small number of coverage elements, so there is no need to report on the multiplicity for each type of elements. Instead, the report presents total multiplicity coverage for all elements, as well as separate coverage values for valid and invalid multiplicity value sets. In order to calculate multiplicity coverage, for each multiplicity value set from the specification, we assign a flag which indicates if this value set appeared in the logs. We then iterate through all multiplicity value sets and count the number of tested valid and invalid multiplicities. Finally, we incorporate this information into the report.

As shown in Figure 3.9, in order to check multiplicity of element e from the specification, for each element of type *e.parent* from logs, we will count the number of elements of type e which appear in the log lines with *e.parent*, and compare this

Figure 3.9 Calculation of components multiplicity coverage

```
For each ComponentType c from spec do
    For each TestCase tc from logs do
       N <- number of components of type c in tc
       For each multiplicity value set mvs of c do
           If (mvs.match(N) = true) then
              mvs.tested <- true
```

number with each of the multiplicity value sets for e defined in the specification. We will mark multiplicity value set as "tested" if it matches the number of elements of type e.

3.4.3.3 Boundary Value Coverage

Boundary value coverage is calculated for value sets which are defined as ranges of numbers. Input and output value sets as well as multiplicity value sets are taken into account. As with multiplicity coverage, it is not necessary to report on the boundary value coverage for each element from the specification; instead, the report contains a total coverage value and a list of not covered boundaries. Calculation of boundary value coverage consists of the following steps:

1. Calculate the total number of boundaries which appear in the specification.
2. Calculate the number of boundary values which appear in the log files.
3. Calculate coverage by dividing the number of covered boundaries by the total number.

Figure 3.10 Calculation of components multiplicity boundary value coverage

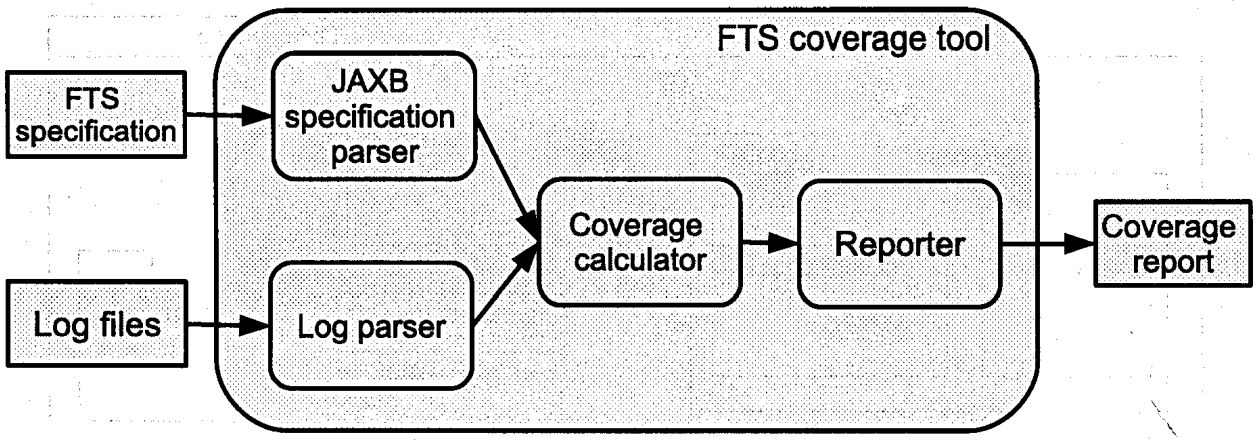
```
For each ComponentType c from spec do
    For each TestCase tc from logs do
        N <- number of components of type c in tc
        For each multiplicity value set mvs of c do
            If (mvs.min != null) then mvs.numBoundaries++
            If (mvs.max != null) then mvs.numBoundaries++
            If (mvs.numBoundaries > 0) then
                If mvs.min = N then mvs.isMinTested <- true
                If mvs.max = N then mvs.isMaxTested <- true
```

In order to calculate the total number of boundaries, we iterate through all value sets which have ranges of numeric values; we add 2 to the total if both min and max values are defined, and add 1 if only one boundary is defined (and the other one is set to"unlimited"). Then, for each value set with the boundary, we check if `min` and `max` values appear in the log file at least once. For example, as shown in Figure 3.10, in order to check the boundary value coverage for component multiplicity value sets, after calculating the number of components of a particular type in a test case, we compare this number with `min` and `max` values for each multiplicity value set which is defined as a range of values, and set appropriate values to `isMinTested` and `isMaxTested` boolean variables.

3.4.4 Architecture

A high-level organization of the FTS Tool utility is presented in Figure 3.11. The program takes as input FTS specification, log files and a report formatting option, and produces a coverage report as a result of its execution. It consists of 4 main components: log parser, specification parser, coverage calculator and reporter. The

Figure 3.11 FTS Tool components

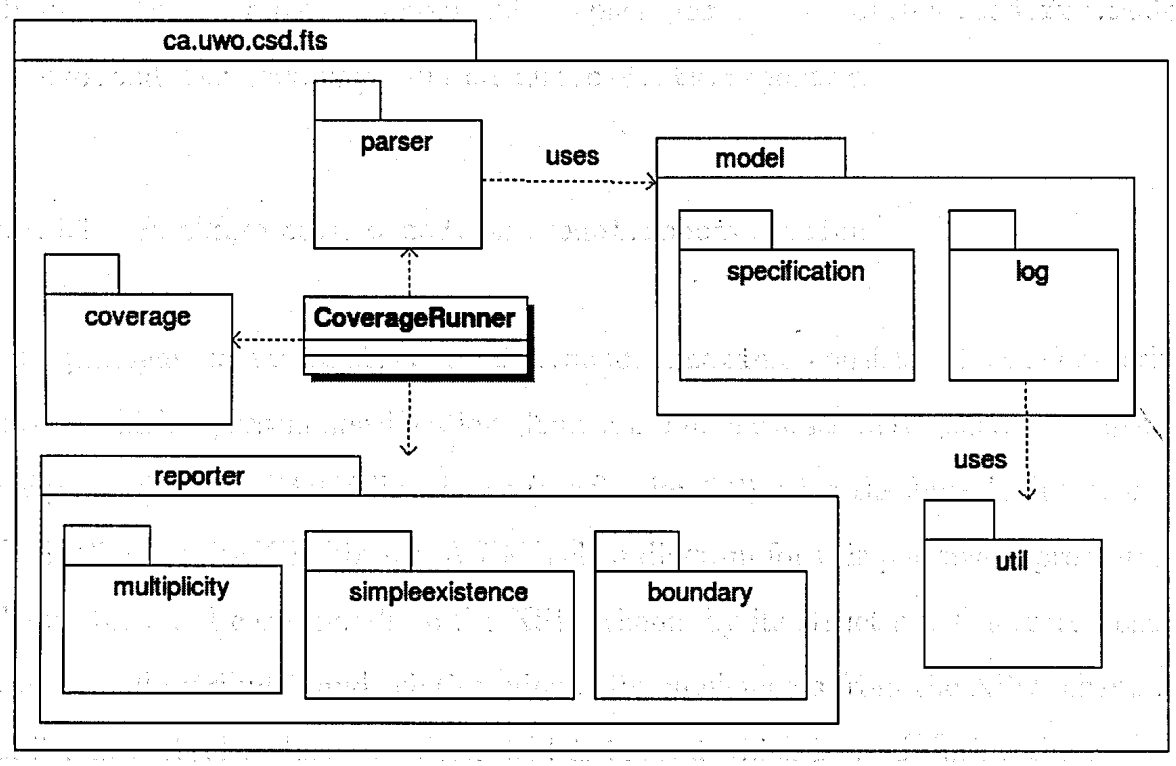


specification parser takes as input an XML file and unmarshalls it into the Java object representation of the FTS. Similarly, the log parser takes as input the location of the log files, and produces the Java object representation of elements which appear in the logs. The coverage calculator iterates through every element from the specification and matches it with elements from log files. As a result, it assigns a "tested" flag to each coverage element from the specification depending on its appearance in the logs. Finally, the reporter component processes all FTS objects, analyzes tested elements, and assembles this information into a report, whose format depends on the option specified by the user.

3.4.5 Design and Implementation

We have carefully designed the FTS Tool in such a way that any modifications and additional functionality could be implemented easily. We use a Unified Modeling Language (UML) package diagram to illustrate package organization of the FTS Tool in Figure 3.12.

Figure 3.12 Package organization of FTS Tool



As shown in Figure 3.12, FTS Tool consists of 11 packages. The `ca.uwo.csd.fts` package contains the main method in the `CoverageRunner` class which is being called when the program is launched. The package `ca.uwo.csd.fts.model` contains two nested packages: `specification` and `log`, which represent the mapping of specification and log elements into Java classes. Classes `LogParser` and `SpecificationParser` from the `ca.uwo.csd.fts.parser` package are responsible for producing log and specification instances correspondingly. The `FTSCoverage` class from the `ca.uwo.csd.fts.coverage` package performs the actual comparison of logs and specification objects. The `ca.uwo.csd.fts.reporter` package contains multiple implementations of the `Reporter` interface which are used to assemble different types of coverage information into a report. Finally, the `ca.uwo.csd.fts.util` package consists

of miscellaneous utility methods. Later in this chapter we explain technical details about the most important packages - `ca.uwo.csd.fts.model`, `ca.uwo.csd.fts.coverage` and `ca.uwo.csd.fts.reporter`.

3.4.5.1 Package `ca.uwo.csd.fts.model.specification`

The package `ca.uwo.csd.fts.model.specification` consists of schema-derived classes which represent specification elements. These classes were generated automatically based on the FTS XSD schema (see Section 3.2) using the Java Architecture for XML Binding (JAXB) library. A UML class diagram for this package is presented in Figure 3.13 and corresponds to the XSD schema by its structure. Generated classes preserve all attributes and relationships between elements from the XSD schema. In addition, each class has a boolean field `isTested` which is used during the coverage calculation. The ValueSetType class has three more additional fields (`isMinTested`, `isMaxTested` and `numBoundaries`), which are used in the boundary value coverage calculation.

3.4.5.2 Package `ca.uwo.csd.fts.model.log`

The package `ca.uwo.csd.fts.model.log` contains Java classes which represent the structure of log files. A UML class diagram for this package is shown in Figure 3.14. The `Log` class is at the top of the hierarchy and represents information about the collection of test cases. It contains a list of `TestCase` objects, each of which has a unique name (canonical file name of the corresponding log file) and a list of `Component` objects. The structure of the `Component` class is similar to the structure of the `ComponentType` class from the FTS. Instead of the Multiplicity property, each

class has a method which returns the number of child elements of a particular type, and this number is used to identify which Multiplicity value sets have been tested. `InputVar` and `OutputVar` classes contain `name` and `value` fields, which are used to identify to which `ValueSetType` they belong.

3.4.5.3 Package `ca.uwo.csd.fts.model.parser`

The package `ca.uwo.csd.fts.model.parser` contains two classes for parsing XML specification and log files correspondingly. The `SpecificationParser` class unmarshalls Java objects using the JAXB library `unmarshal` method, and returns an instance of `SystemDescType` class, the root specification element.

The `LogParser` implements the functionality of parsing log files and producing instances of the `ca.uwo.csd.fts.model.log` package classes. In contrast with the `SpecificationParser`, the implementation is more complex, because log files have a custom structure and can not be parsed automatically with the help of an external tool. The parser reads log files line by line and creates corresponding objects. `LogParser` also checks log files for validity. First, each line should start with an appropriate keyword, which identifies the type of element. Second, each element which is referenced in the log file, must be defined earlier in the same file. For example, input source cannot be defined before the component to which it belongs. `LogParser` should also take into account the fact that if a value of input or output variable contains whitespaces, it is surrounded by double-quotes, and any double-quote should be preceded by the backslash.

3.4.5.4 Package `ca.uwo.csd.fts.model.coverage`

The package `ca.uwo.csd.fts.model.coverage` contains class `FTSCoverage` which is responsible for the coverage calculation. `FTSCoverage` class iterates through the `ca.uwo.csd.fts.model.specification` and for each object checks if the element of this type appeared in the `ca.uwo.csd.fts.model.log` structure. Coverage calculation for each component type is performed according to the algorithm described in Section 3.4.3.

3.4.5.5 Package `ca.uwo.csd.fts.model.reporter`

The package `ca.uwo.csd.fts.model.reporter` contains classes for producing the coverage report. UML class diagram for this package is shown in Figure 3.15. The final report, which contains coverage information about all component types, can be broken down into small parts, each representing a particular coverage aspect. Similarly, we have decided to break down the functionality of producing the report. This will allow us to easily modify any existing part of the report, or add a new section to it. We have created the `Reporter` interface with the `report` method, which is implemented by all of the reporter classes. In order to produce a report, the `ReportFactory` class creates instances of the `Reporter` interface, and then the `CoverageRunner` calls the `report` method on all reporter instances.

Classes `SimpleExistenceReporter`, `MultiplicityReporter` and `BoundaryReporter` calculate the total coverage of the corresponding types, while other classes, such as `ComponentsReporter`, `InputSourcesReporter` and others, print coverage of particular element types. These classes use similar algorithm - iterate through every FTS object, count the number of tested elements and divide it by the total

number of elements of the specified type. These classes are inherited from the abstract class `ReporterImpl`, which implements the `Reporter` interface. Classes `SimpleExistenceMissingReporter`, `MultiplicityMissingReporter` and `BoundaryMissingReporter` are used to produce the second section of the report, where missing coverage elements are printed. `NotMatchedVarsReporter` is a separate type of a reporter, because its `report` method has an additional `Log` parameter. In contrast with the coverage calculation, this class iterates through the instances of the logs, and checks, if every input and output variable value was assigned to a particular value set from the specification. If it finds a value which does not match any of the value sets from the specification, this value is included in the report. Classes `SimpleExistenceCEReporter`, `MultiplicityCEReporter` and `BoundaryCEReporter` are used when the `-listce` report format option is specified by the user, and the list of unique identifiers of each coverage element should be printed. `ReportWriter` is a helper class, which implements the `write` method, and is responsible for creating a new report file with the unique name in the `reports` folder.

Figure 3.13 Class diagram of `ca.uwo.csd.fts.specification` package

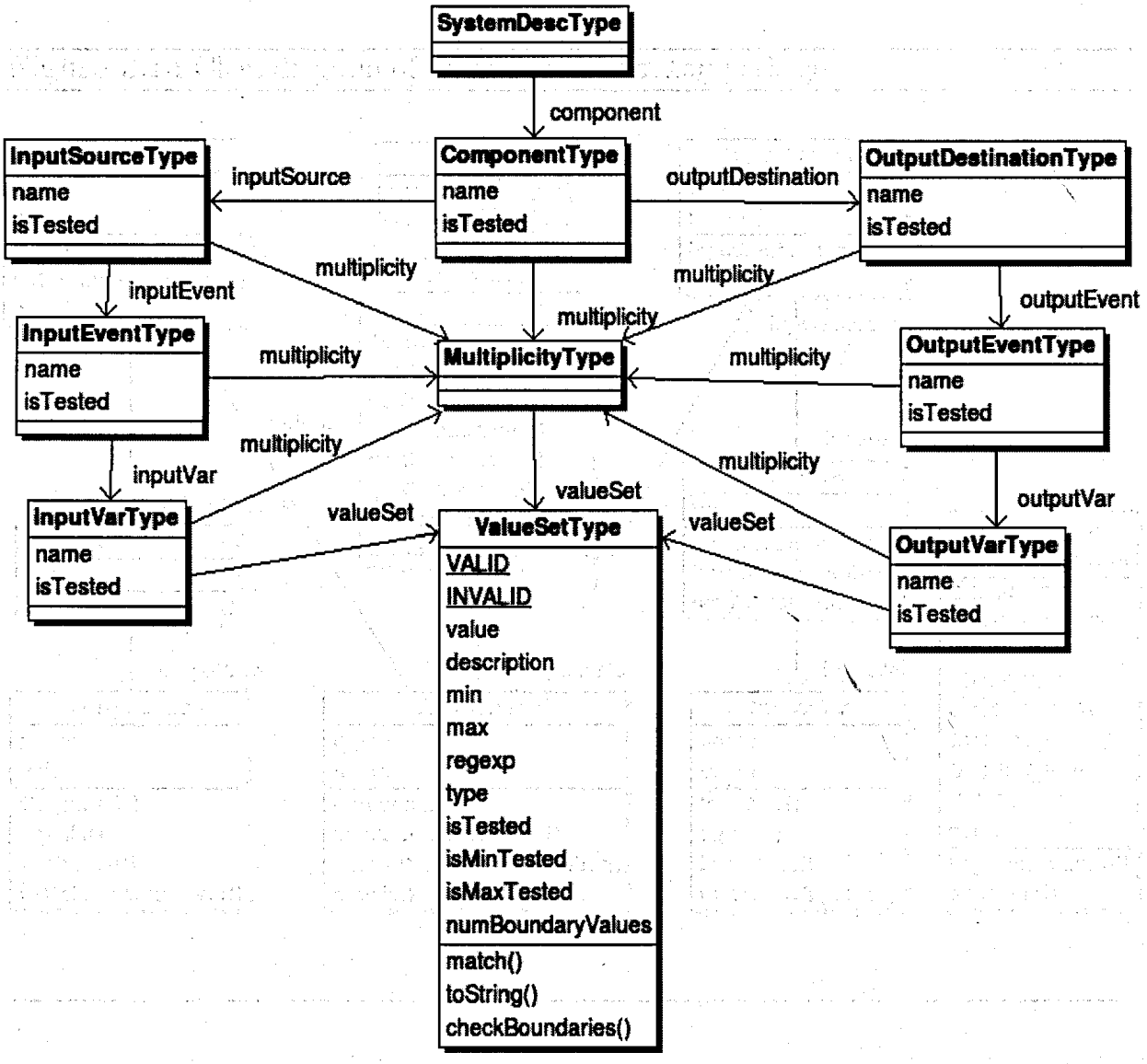


Figure 3.14 Class diagram of `ca.uwo.csd.fts.log` package

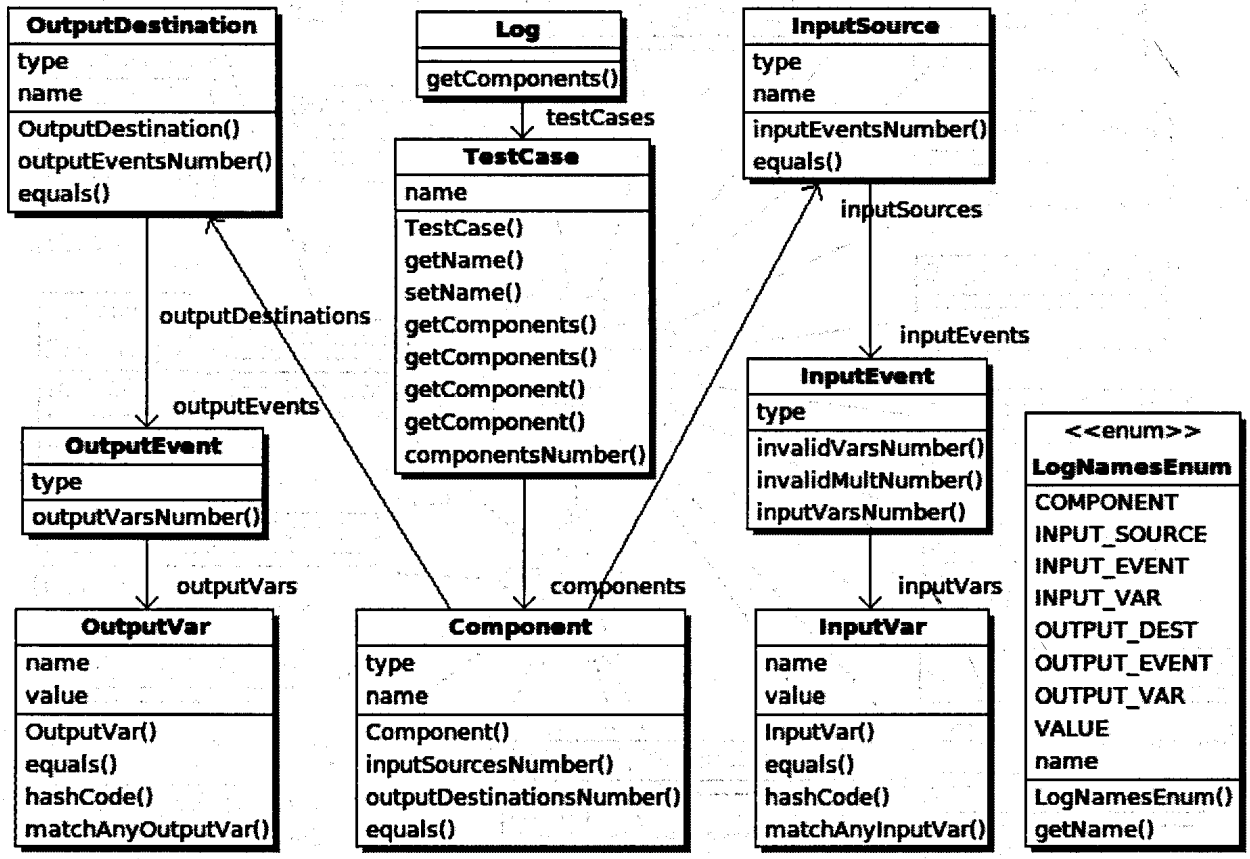
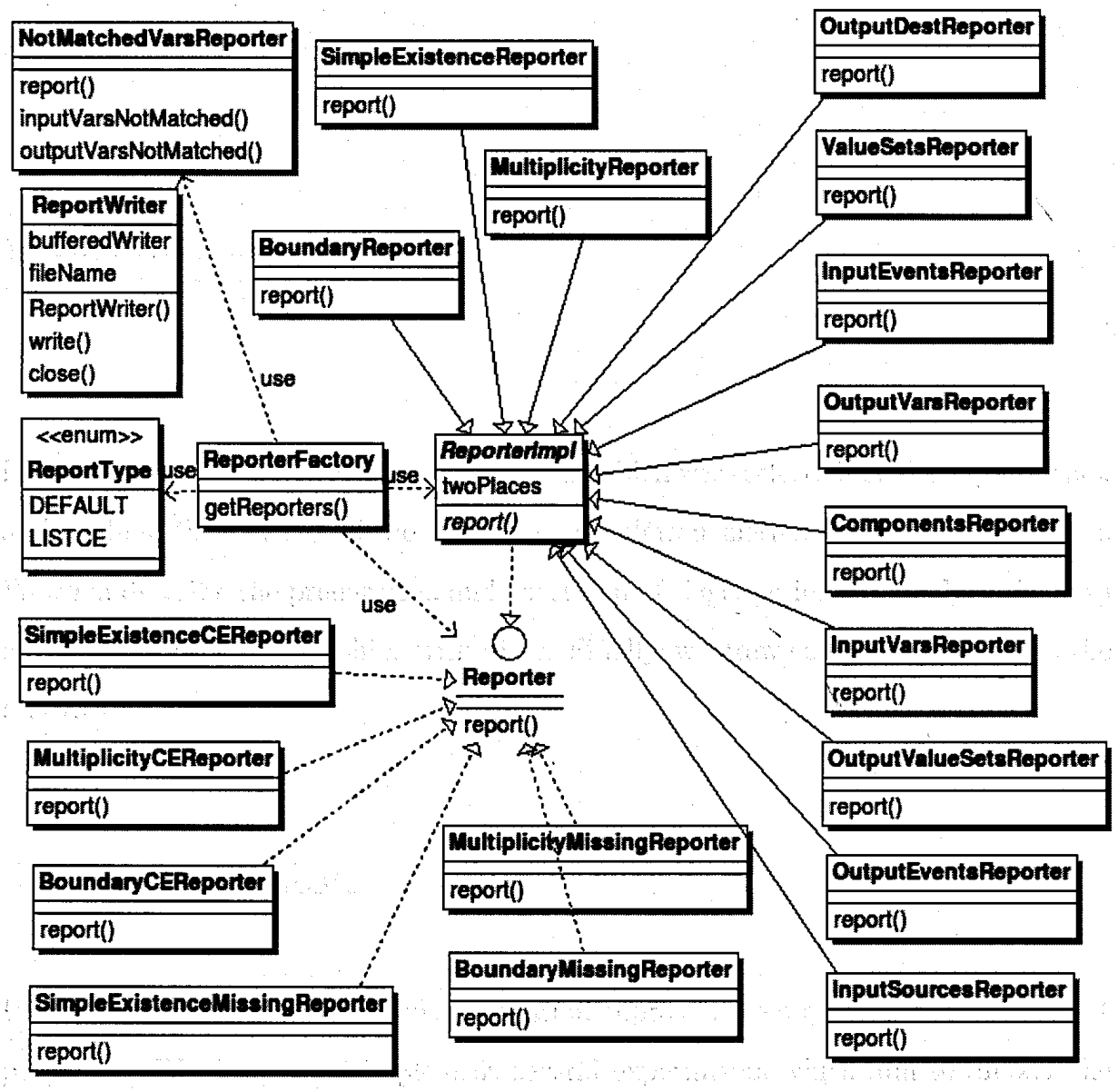


Figure 3.15 Class diagram of `ca.uwo.csd.fts.reporter` package



Chapter 4

Experiments

In this chapter, we describe subject programs which were selected for the experiments, and evaluate how the black box coverage calculation method was applied to them. We then describe the preparation and execution of the experiments, analyze the data and illustrate the relationships with plots. Finally, we draw conclusions based on the analysis.

4.1 Motivation

In order to evaluate the black box calculation approach, we apply it to several subject programs. We design and implement several experiments which aim to answer the following research questions:

- How easy is it to apply the black box coverage calculation approach to subject programs of different sizes and programming languages?

Table 4.1 Characteristics of subject programs

Program	Language	Number of test cases	SLOC	Classes	Functions
`flex`	C	567	10,421	N/A	162
`concordance`	C++	372	1,034	5	39
`yamm`	Java	238	780	13	48

- What is the relationship between the black box and white box coverage measures?
- What is the relationship between the black box, white box, test suite size and effectiveness? Is it consistent with the experimental results in [25]?
- Is the black box coverage a good predictor of test case effectiveness?

4.2 Subject Programs

In order to test our approach on programs of various sizes, functionality and programming languages, we have selected the following subject programs for our experiments: `flex`, `concordance` and `yamm`. Characteristics of these programs can be found in Table 4.1. The size of programs was estimated using the SLOC (lines of code not counting comments or whitespace) metric, which was calculated by the LLOC tool [22]. `flex` is a C program with the biggest size and the largest test case pool; `concordance` is a medium-size utility C++ program; and `yamm` is a Java GUI-based client-server program.

4.2.1 Subject Program `flex`

The subject program `flex` (fast lexical analyzer generator) is a free implementation of the original Unix `lex` program, which generates programs that perform pattern-matching on text. The program takes as input a text file which consists of three sections: definitions, rules and code. After processing an input file `flex` generates a C source code file "lex.yy.c" which implements the `yylex()` function. The *rules* section specifies pairs of regular expressions and C code, such that when the "lex.yy.c" file is compiled and run on some input, it analyzes input text, and executes the corresponding C code each time it finds a text which matches a regular expression defined in the *rules*. The *definitions* section of an input file can be used to ease the construction of rules by assigning custom names to regular expressions used in the *rules* section. The last optional section of the input file contains custom C code which is copied to "lex.yy.c" without any modifications.

In our experiments we use the version of `flex` which was obtained from the Software-artifact Infrastructure Repository (SIR) at the University of Nebraska-Lincoln. The package contains several sequential previously released versions of the program, and we use the latest version `v5` for our experiments. The SIR researchers have used the informal documentation to create a specification in the TSL language [9]. After applying a TSL generator to it they have obtained textual descriptions of test cases in the form of TSL test frames, and assigned a line in a "universe" format to each test frame. The package which was obtained from SIR contained 6 TSL test frames with "universe lines" assigned to them; in order to get an automated shell script which could execute all test cases, we ran a JavaMTS tool (also obtained from SIR) with a "universe" file, which produced an automated test suite consisting of 567 test cases.

4.2.2 Subject Program concordance

The subject program `concordance` is a C++ utility program which makes word indices of documents. It was introduced as a subject program for the first time by Andrews et al. and is described in [25]. This program takes as input a text file "filename" and creates two output files with the information about concordances. The output file "filename.wds" contains a list of all words used in the input file. It also specifies the length of each word, number of times it appeared in the input file, and locations where it appeared. Location can be counted by a page, line or stanza, and can be specified using options `p`, `l` or `s`. The output file "filename.abc" contains a list of characters from the input file, the number of appearances and the overall percentage of uses for each character. In our experiments, we use the pool of 372 test cases created for the study in [25].

4.2.3 Subject Program yamm

The subject program `yamm` (Yet Another Mastermind) is a version of a famous mastermind game written in Java. It is a multi-player version of a game with a GUI implemented using the Java AWT library. This program consists of two components: server and client. In order to play the game, the server should be launched first, with the port number and the mode of generating game combinations specified. Then one or more clients can connect to the server, using the server address and port number, and specifying the user name. When the connection is established, a GUI is shown to the user, where he can select colours by clicking on circles, and submit his guess. The guess is evaluated by the server, and the response is sent back, which tells the user how close he is to the winning combination. The first player who guesses correctly

within 12 attempts, wins the game.

This program was written by Laurent Cavallin and Corentin Massot and can be found on various open source repositories in the Internet. The version which is being used is a beta release 0.10beta2. Test cases for this subject program have been created by 22 students in the class of Computer Science 4472 at UWO in 2009. The Abbot automated testing package [1] was used in test cases to access and manipulate GUI components of `yamm`. These test suites formed a test pool with the total of 256 test cases.

4.3 Evaluating Black Box Coverage Calculation Method

In order to evaluate the black box coverage calculation method, we have executed the following steps for each of the subject programs:

- Analyze the informal specification and perform equivalence partitioning on inputs and outputs.
- Assemble equivalence classes into an FTS XML specification.
- Instrument the source code so that logs will be written in the specified format.
- Create a bash shell script which will execute all test cases automatically.
- Execute the script and collect log files.
- Run FTS Tool with the FTS specification and log files in order to obtain the coverage report.

We were able to successfully apply the black box coverage calculation to all subject programs. However, `flex` appeared to be the most difficult program to create an FTS specification for, primarily because of the large number of switch options which could be specified as input for `flex` and affect the generation of the "lex.yy.c" file. According to the informal program specification, it is not enough to test each switch individually; instead, there is a need to run the program with certain combinations of switches and also check that invalid combinations are tested as well. Therefore there is a need to take into account valid and invalid combinations of value sets, which is not supported by the current implementation of the FTS Tool. This limitation of our approach is discussed in the next chapter. Figure 4.1 shows an extract from the FTS specification for the `yamm` subject program. It presents the FTS specification in a plain text format, omitting closing tags, angle brackets and `Multiplicity` and `ValueSet` elements. As an example, `ValueSet` elements for the *PortNumber* input variable have been included.

Instrumenting the `flex` source code also appeared to be more difficult than instrumenting the two other programs. It was required by the specification that the contents of *rules* and *definitions* sections of the input file are captured. However, it was not possible to capture these values in the logical flow of the program which was implemented in a state machine fashion, where each character of an input file was processed individually. As a result, we had to add code which would parse the input file, extract the necessary information and write it into the log file.

After running all available test cases and obtaining execution logs, we have calculated the black box coverage for each of the subject programs. Table 4.2 provides coverage values for each of the subject programs displayed both as a ratio of the number of covered elements to the total number of elements, and as a percentage. As shown in

Figure 4.1 FTS specification for yamm

```
SystemDesc
  Component name=Server
    InputSource name=CommandLine
      InputEvent name=ServerInitialized
        InputVar name=PortNumber
          Valid ValueSet From 1024 to 65535
          Invalid ValueSet Negative numbers
          Invalid ValueSet From 0 to 1023
          Invalid ValueSet From 65536 to unlimited
          Invalid ValueSet Number in incorrect format
        InputVar name=GameMode
    InputSource name=StdInput
      InputEvent name=GameInitialized
        InputVar name=ColoursSpecified
        InputVar name=PegColour
    InputSource name=ComboGenerator
      InputEvent name=GameInitialized
    OutputDestination name=ClientConnection
      OutputEvent name=UserWon
        OutputVar name=NumberOfGuesses
      OutputEvent name=GameFinished
        OutputVar name=Result
      OutputEvent name=GuessEvaluated
        OutputVar name=NumberOfRedLines
        OutputVar name=NumberOfWhiteLines
  Component name=Client
    InputSource name=ServerMessage
      InputEvent name=NewGameMessageReceived
      InputEvent name=GameOverMessageReceived
      InputEvent name=ConnectionSucceededMessageReceived
      InputEvent name=ConnectionFailedMessageReceived
    InputSource name=CommandLine
      InputEvent name=ClientInitialized
        InputVar name=PortNumber
        InputVar name=UserName
        InputVar name=ServerName
    InputSource name=MastermindGUI
      InputEvent name=UserGuess
        InputVar name=ColoursSelected
      InputEvent name=GameStopped
      InputEvent name=ClickOnPeg
        InputVar name=PegNumber
        InputVar name=NumberOfClicks
```

Table 4.2 Black box and white box coverage measurements for subject programs

Coverage elements	`flex`	`concordance`	`yamm`
Simple existence	125/139 = 89.93%	74/79 = 93.67%	100/106 = 94.34%
Valid value sets	59/66	28/28	35/38
Invalid value sets	12/14	17/21	11/14
Output value sets	12/17	2/2	15/15
Multipl. value sets	50/72	12/12	19/23
Boundary value sets	3/48	5/12	13/32
Total black box	178/259 = 68.73%	91/103 = 88.35%	132/161 = 81.99%
Total white box	80.73%	100%	91.76%

the table, the `yamm` subject program has the highest simple existence coverage, while concordance has the highest total coverage. All three programs have low boundary value coverage, but as `concordance` has the smallest number of boundary values in the specification, its total coverage is the largest among the three programs. Low boundary value and invalid value sets coverage can be explained by the fact that boundary and incorrect conditions are usually taken into account after all valid test cases have been explored, and very often these conditions are forgotten. As we can see from Table 4.2, the total number of coverage elements appears to be greater than 100 for all subject programs, however, it still provides less granularity than the white box statement coverage in which the number of coverage elements equals to the number of executable lines of code. We can also see from Table 4.2 that `flex` has the smallest white box coverage value among the three programs, while `concordance` has the highest coverage value of 100%.

4.4 Comparison of Black Box and White Box Coverage Metrics

The first set of experiments which we execute aims to compare the white box coverage of each single test case with its black box coverage.

4.4.1 Experiment Design

In the first step of the experiment we obtain the white box coverage using the `gcov` utility for `flex` and `concordance`, and the `Cobertura` utility for `yamm`. The basic principles of statement coverage calculation utilities is described in Section 1.2.1. The first step is to compile the original source code of each subject program using two special GCC options `-fprofile-arcs -ftest-coverage` or instrument Java bytecode with `Cobertura` after compiling `yamm` source code. After execution of each test case we generate a coverage report, save it and clear the coverage file for future use. As the statement coverage report is generated in the format specific to the coverage calculation tool, the next step is to transform a report generated for each test case into a plain text file which contains only numbers of lines executed by a particular test case. In case of a `Cobertura` report, each line contains a class name and a line number. Finally, we calculate the white box statement coverage by dividing the number of unique lines in each file by the total number of coverable source lines.

The second step is to collect the black box coverage information using the FTS Tool. We run the FTS Tool for each individual log file with the `-listce` option, which produces a report file with the list of unique identifiers of covered elements. As we can see from Table 4.2, none of the test suites for subject programs achieve 100%

black box coverage, and only `concordance` achieves 100% white box coverage. For the comparison purposes, we calculate the relative coverage values, where the maximum coverage achieved by the whole test pool will be considered as a 100%. Finally, we assemble percentage values for each of test cases into a single text file and analyze them.

In order to see if combining individual test cases into test suites will have an effect on the relationships which we study, we generate test suites of various sizes from 2 to 50, with 100 test cases in each test suite, resulting in 4900 test suites in total. For each test suite, we choose test cases randomly using a permutation tool, and store test case numbers in a text file corresponding to this test suite. As we have information about black box and white box elements covered by each test case, we are able to calculate the coverage of each test suite without actually running test cases or running the FTS Tool on the log files. Instead, we compute the union of the collections of coverage elements of individual test cases which comprise this test suite, and divide the number of distinct elements in the set by the total number of coverage elements.

4.4.2 Experimental Results

Figure 4.2 shows the scatterplot of the total black box and white box statement coverage measures for individual test cases of `flex`. As we can see from the figure, the majority of the test cases are grouped together and have a positive correlation between the two coverage measures. The white box coverage measure is greater than the black box measure for all test cases in this group. There is also a small number of outliers for which the black box coverage is greater than the white box coverage, and the white box coverage is smaller than the average white box coverage. These outliers

Figure 4.2 Scatterplot of black box and white box measures for `flex`

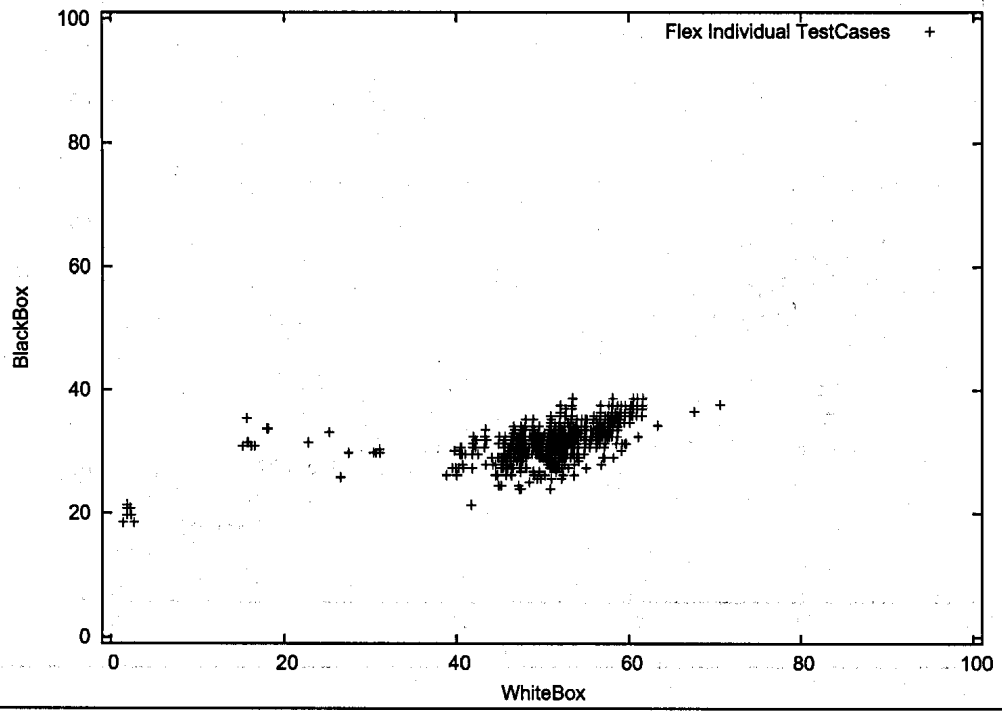


Figure 4.3 Scatterplot of black box and white box measures for `concordance`

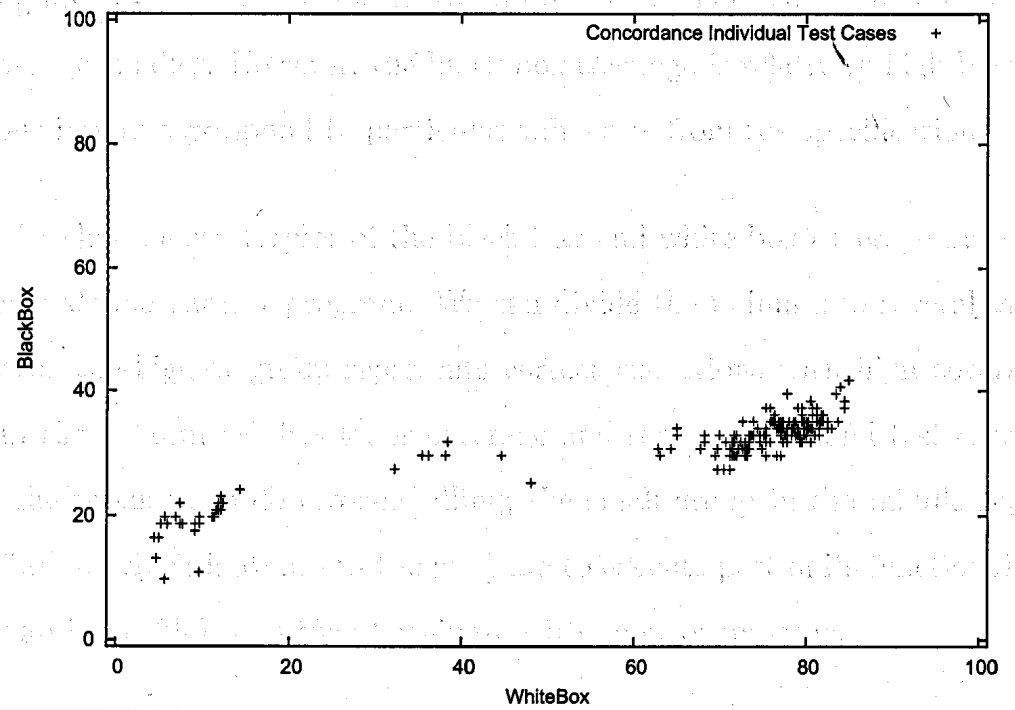
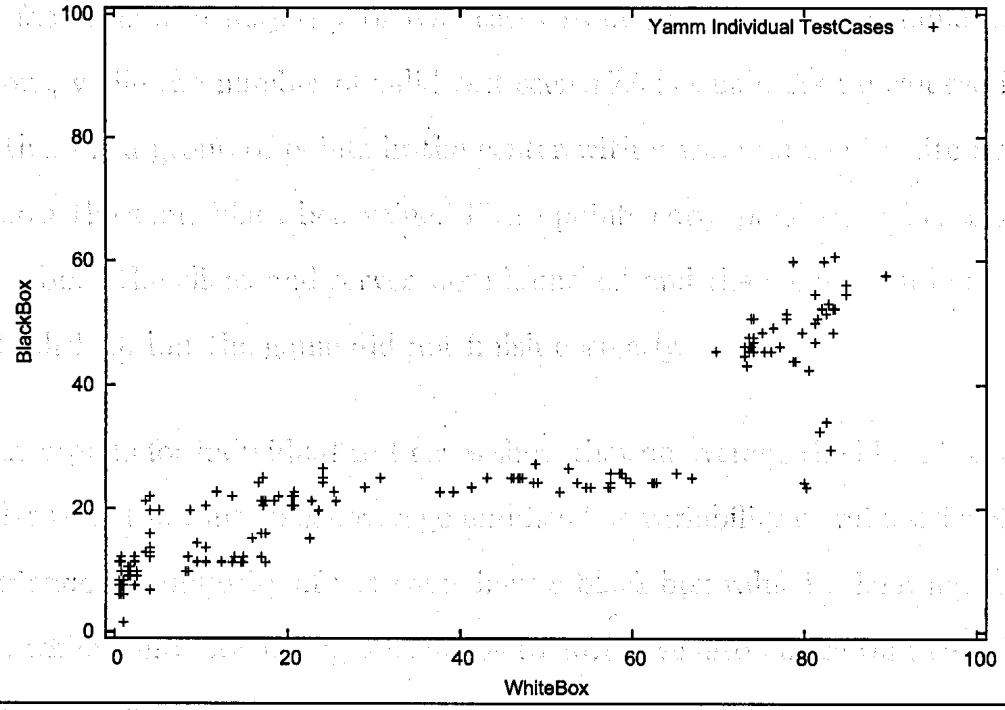


Figure 4.4 Scatterplot of black box and white box measures for `yamm`



represent erroneous test cases which test invalid inputs to the program. For these test cases, only the error-handling lines of code are executed. The number of the error-handling lines is small compared to the total size of the program, making the white box coverage smaller. However, the black box coverage is relatively high because the erroneous inputs correspond to particular value sets from the specification.

Figure 4.3 shows a scatterplot of the black box and white box coverage measures for the `concordance` subject program. We can divide the points into several groups of test cases: the biggest group represents correct executions with high coverage; the group in the bottom left has lower coverage and represents invalid test cases which caused the program to do error-handling; the small group in the middle represents invalid inputs which have forced the program to execute part of its functionality (e.g. parsing an input file), and then terminate with an error message.

Figure 4.4 shows a scatterplot for the `yamm` program. In this scatterplot, the number of

erroneous test cases with small coverage values is rather large. This can be explained by the fact that the majority of test cases from the test pool examine erroneous conditions, while the number of valid test scenarios is small. As we can see from the figure, there is a group of points in the center with a wide range of white box values but almost the same black box value. These points correspond to test case scenarios in which both the client and server were launched and the connection between then was established, but the game did not finish correctly.

The scatterplots for individual test cases show that on average the black box coverage is smaller than the white box coverage and has less variability of values: for `flex` and `concordance` the majority of test cases have a black box value in the range from 20% to 40%, while white box changes from 0% to 80%. We also notice that the black box coverage value does not directly correspond to the number of executed lines of code. Consider two test cases: the first one tests an erroneous condition, which causes the program to execute only the error-handling code; while the other test case tests a valid input to the program, which causes the program to execute its main functionality. These two test cases have similar black box coverage values but the first one has low white box coverage, and the second one - high white box coverage.

We also analyze the relationship between black box and white box coverage measurements for the pool of randomly generated test suites of sizes from 2 to 50, containing 4900 test suites in total. Figures 4.5, 4.6 and 4.7 show scatterplots of the total black box and white box statement coverage measures for the `yamm`, `concordance` and `flex` subject programs respectively. In these plots, each point corresponds to one test suite from the pool of test suites of sizes from 2 to 50. As we can see from these figures, all `flex` test suites are grouped together, while the `concordance` and `yamm` plots have a group of outliers with smaller coverage values. This can be explained by the fact

Figure 4.5 Scatterplot of black box and white box measures for `yamm` test suites

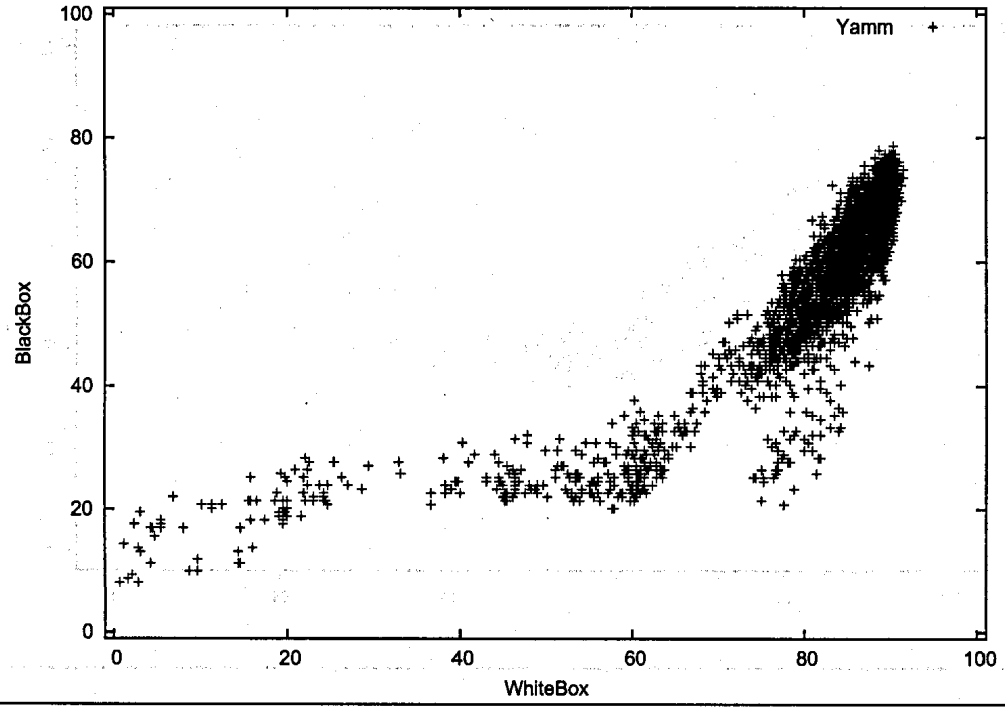


Figure 4.6 Scatterplot of black box and white box measures for `concordance` test suites

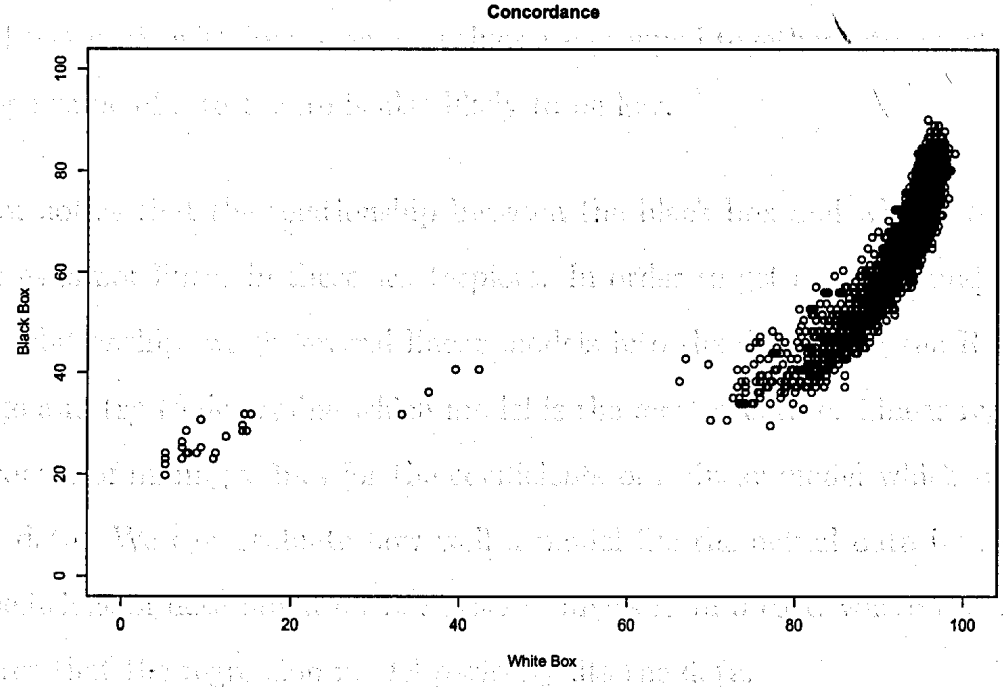
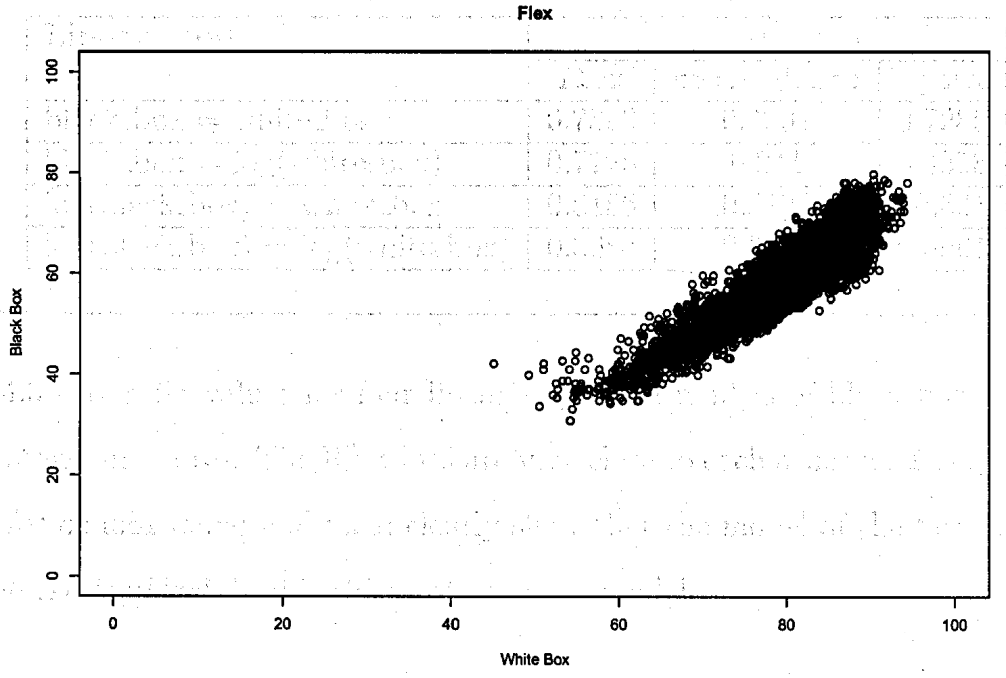


Figure 4.7 Scatterplot of black box and white box measures for `flex` test suites



that in the scatterplots with black box and white box values for individual test cases, the number of outliers with smaller black box and white box values is considerably smaller for `flex` in comparison with `concordance` and `yamm`. Correspondingly, when several test cases with low coverage values are grouped together into a test suite, the coverage value of a test suite is also likely to be low.

We can notice that the relationship between the black box and white box coverage measures is not linear in these scatterplots. In order to get a deeper understanding of the relationship, we fit several linear models into the data using the R statistical package and try to determine which model is the most accurate. Linear regression is the process of finding values for the coefficients of a linear model which best fit the actual data. We can evaluate how well a model fits the actual data by calculating the coefficient of determination R^2. An R^2 ranges from 0 to 1, where the value of 1 indicates that the regression model perfectly fits the data.

Table 4.3 Goodness of fit of black box vs. white box relationship, measured by R^2

Linear model	**R^2**		
	`flex`	`concordance`	`yamm`
black_box = white_box	0.7899	0.4997	0.7289
black_box = log(white_box)	0.7725	0.241	0.4518
log(black_box) = white_box	0.8156	0.6122	0.831
log(black_box) = log(white_box)	0.8084	0.3393	0.5961

Table 4.3 shows R^2 values for four linear regression models of black box and white box coverage measures. The R^2 values are very close to each other for `flex`, while the values for `concordance` and `yamm` clearly show that the model of the form *white_box = B0 + B1*log(black_box)* is the most accurate model.

We have noticed from the scatterplots for individual test cases that there exist a lot of test cases with relatively low black box coverage which execute most of the program's code. When test cases are combined into test suites, we can see a large number of test suites with low black box coverage and a relatively high white box coverage. However, if we look at test suites with higher black box values, we can see that when the black box coverage of a test suite increases, its white box coverage does not change significantly.

4.5 The Relationship Among Size, Coverage and Effectiveness

Andrews' experiments [25] indicate that both size and coverage influence test suite effectiveness, and a linear relationship among variables log(size), coverage and effectiveness exists for studied programs. In this set of experiments we determine if adding

the black box coverage to the model makes it more accurate, and if a nonlinear relationship among the black box coverage, size and effectiveness of a test suite still holds.

We use two subject programs `flex` and `concordance` for this experiment, as existing test cases for `yamm` have limitations which make this program not suitable for the experiments. Specifically, `yamm` test cases do not have a mechanism of checking if a mutant was detected by a particular test case. The majority of `yamm` test cases interact with the client GUI using the Abbot library, and do not have a mechanism to check if all elements of GUI are in the correct state.

4.5.1 Mutant Generation

The first step of the experiment is the preparation of faulty versions of the original programs. We reuse `concordance` mutants which were generated for studies in [25]. We generate mutants for `flex` using the mutant generator which is described in [6], and which uses four types of mutant operators: "replace operator", "replace constant", "negate decision" and "delete statement". We apply mutant operators to the lines of code which were covered by the test pool. We then identify equivalent and non-equivalent mutants. In order to do this we first run test cases on the "gold" version of the program (the original version with no known faults) and save that program's output, which will act as a test oracle. We then run all test cases on each of the faulty versions, and consider a mutant to be equivalent if the output of all test cases is the same. If any test case produces a different output while being executed on the faulty version, this version is considered to be non-equivalent. The ratio of non-equivalent to equivalent mutants appeared to be different for the two subject

programs: we have selected 135 non-equivalent mutants for `concordance` from the pool of 200 mutants, and 125 non-equivalent mutants for `flex` from the pool of 1000 mutants. In this experiment, the mutant equivalence is approximated because we are using only a subset of test cases from the infinite set of potential test cases, in order to decide if a mutant is equivalent. We have prepared a set of shell scripts to automate the execution of subject programs and collection of the experimental data in order to make the experiments reproducible.

4.5.2 Data Collection

The best way to measure how size, black box and white box coverage separately influence the effectiveness is to consider low and high values of each factor: (low size, high size), (low black box, high black box), (low white box, high white box). In order to use this approach, we would have to construct test suites with all combinations of factor levels, measure the effectiveness of each test suite, and then perform the 4-factor 2-level analysis of variance (ANOVA) to see which factors influence the effectiveness. However, our initial analysis showed that all three factors - size, black box and white box coverage, are positively correlated, and constructing a test suite with high black box and white box coverage but small size is almost impossible. Instead we perform the analysis of covariance (ANCOVA) which requires a continuous outcome variable, at least one categorical factor variable and at least one continuous factor variable, and combines features of simple linear regression with ANOVA.

Therefore we use a data set which consists of 100 random test suites of each size from 2 to 50, resulting in 4900 test suites. Generation of this set of test suites was described in Section 4.4.1. For each test suite, we record the black box and white box coverage

as well as the effectiveness, which is calculated as a fraction of the number of mutants killed by a particular test suite. In this data set, we have one continuous outcome variable (effectiveness), two continuous factors (white box and black box coverage), and one discrete factor (size).

As the way of generating test suites might affect the result, we also prepared a set of test suites which achieve particular black box coverage thresholds, and generated 1000 test suites of each black box threshold value. We picked thresholds 50, 60, 70, 80, 85, 90, 91, 92 and 93 for `concordance`, and 50, 60, 70, 75, 80, 81 and 82 for `flex`, as the maximum feasible coverage of the `flex` and `concordance` test pools is 82% and 93% correspondingly. The earlier studies by Andrews et al. [8] have shown that the effectiveness rises sharply as the 100% feasible white box coverage is approached. In order to see if this pattern holds for the black box coverage, we made the threshold black box coverage values more fine-grained as we approached the maximum feasible coverage. We did not add any test cases to the existing test pools to achieve 100% black box coverage, as we did not want to change the test pools that were supplied with the subject programs. For each test suite, we record the white box coverage, test suite size and effectiveness. In this case, a discrete factor is the black box threshold, and two continuous variables are size and white box coverage.

4.5.3 Experimental Results

We visualize the experimental data with various plots, perform ANCOVA in order to see which factors influence the effectiveness, and perform several linear regressions in order to get a deeper understanding of the relationships among factors. We use the R statistical package [29] for statistical analysis of data.

4.5.3.1 Visualizations

We first analyze test suites of fixed sizes from 2 to 50, and compare their effectiveness, black box and white box coverage values with their sizes. Figure 4.8 shows how test suite size influences the effectiveness, for the `concordance` subject program. This figure shows a box and whisker plot with distributions of the effectiveness values for each test suite size. As we can see from the figure, test suites of sizes from 2 to 6 have several outliers, which have low effectiveness values. These outliers correspond to test suites which consist of test cases with very low white box and black box coverage measures. Effectiveness increases slowly for test suites of sizes from 7 to 30, and has very similar distributions and mean values for test suites of sizes from 31 to 50.

Figures 4.9, 4.10 and 4.11 show how test suite size influences the effectiveness, white box and black box coverage, for both subject programs. In all three figures, each point represents the average value for 100 test suites of the given size. All three figures show a positive correlation among the factors.

Figures 4.12 and 4.13 show scatterplots of the black box coverage and test suite effectiveness for the `concordance` and `flex` subject programs respectively. In these plots, each point corresponds to one test suite from the pool of test suites of sizes from 2 to 50. These scatterplots show that increasing the black box coverage from 35% to 55% results in the significant increase of effectiveness for both subject programs, while increasing the black box coverage from 70% to 90% does not result in the increased effectiveness.

We also analyze the set of test suites which have fixed black box threshold values. Figures 4.14, 4.15 and 4.16 show how black box coverage measure influences the effectiveness, white box coverage and size, for both subject programs. In all three

Figure 4.8 Boxplot of the effectiveness and size for `concordance` test suites

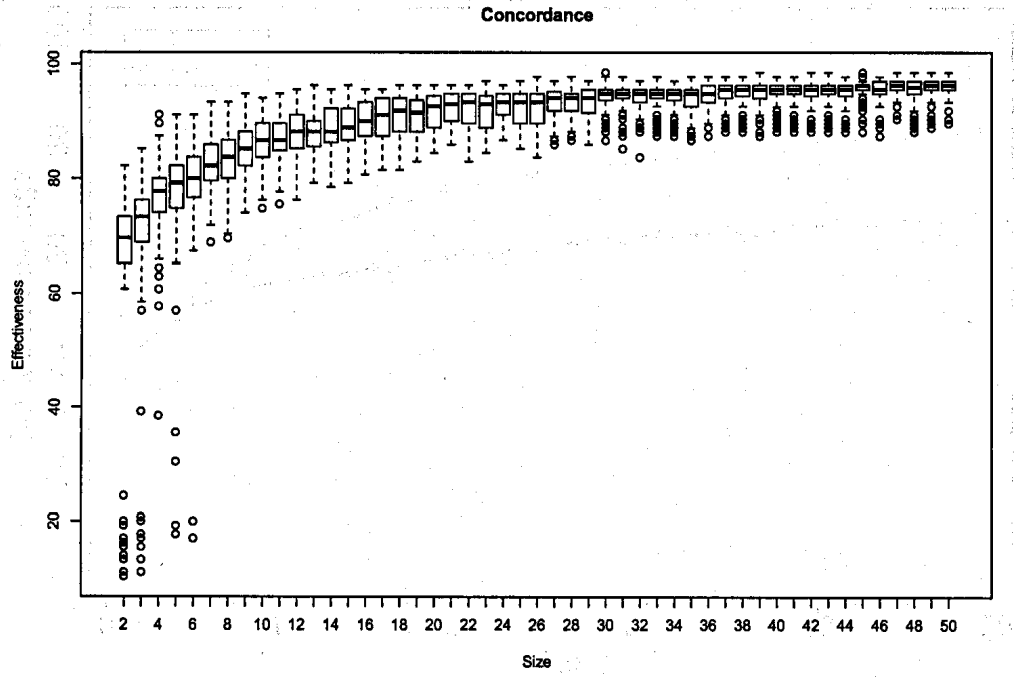


Figure 4.9 The relationship between size and effectiveness

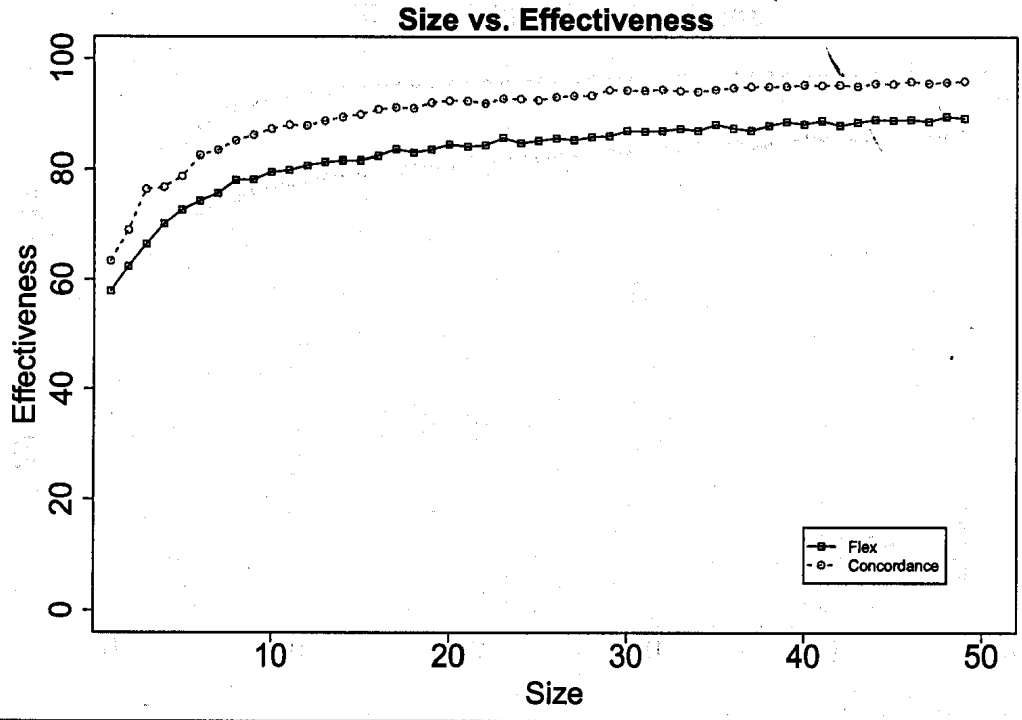


Figure 4.10 The relationship between size and black box coverage

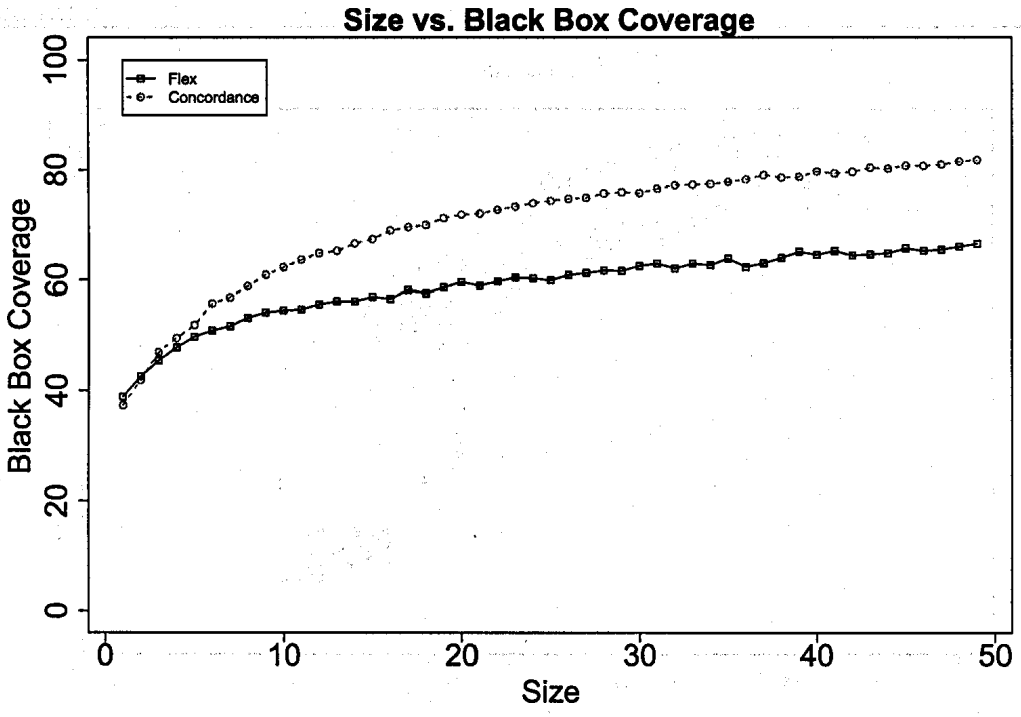


Figure 4.11 The relationship between size and white box coverage

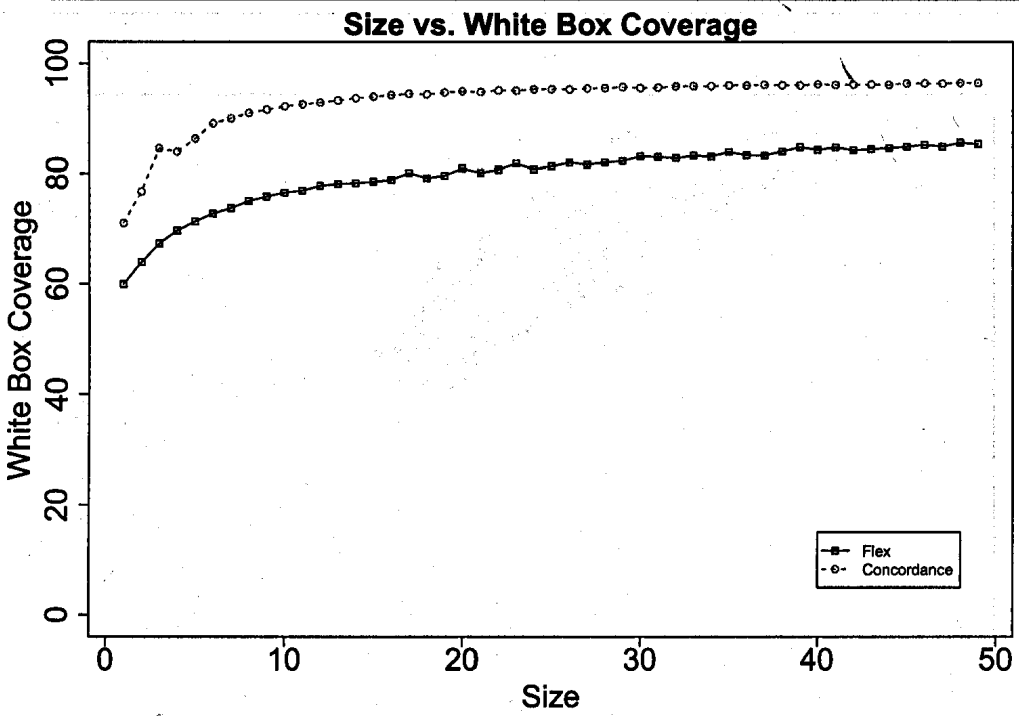


Figure 4.12 The relationship between black box coverage and effectiveness for `concordance`

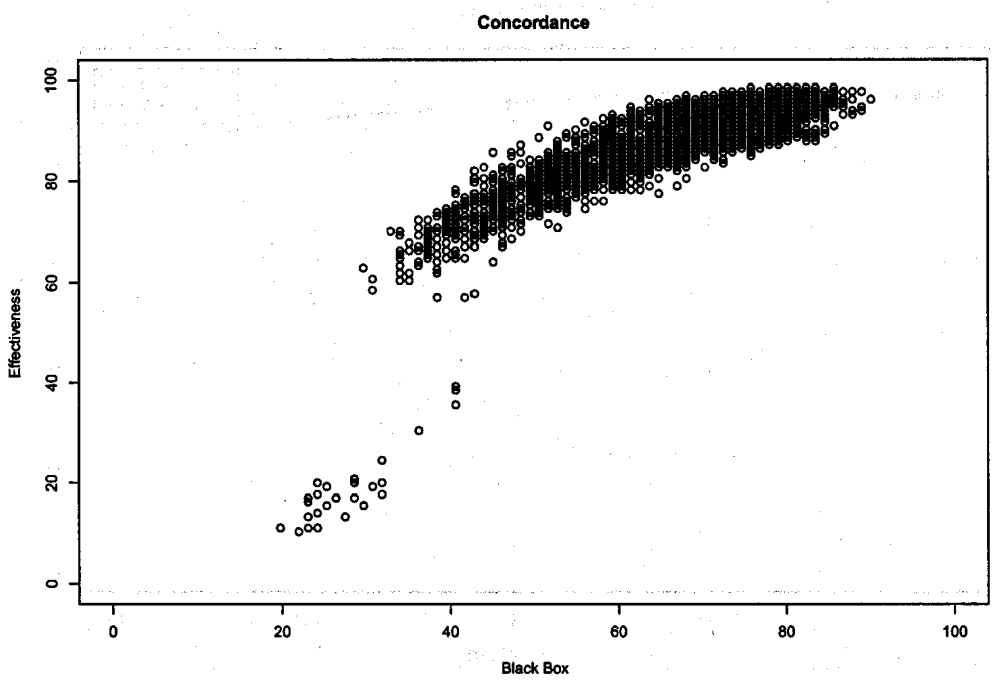


Figure 4.13 The relationship between black box coverage and effectiveness for `flex`

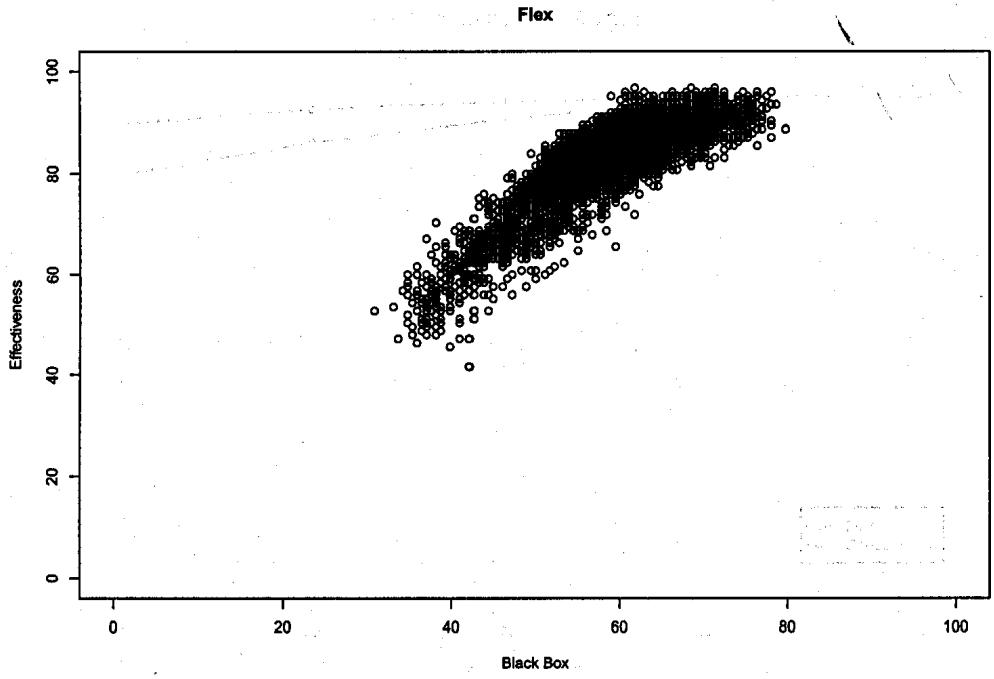


Figure 4.14 The relationship between black box coverage and effectiveness for discrete black box data set

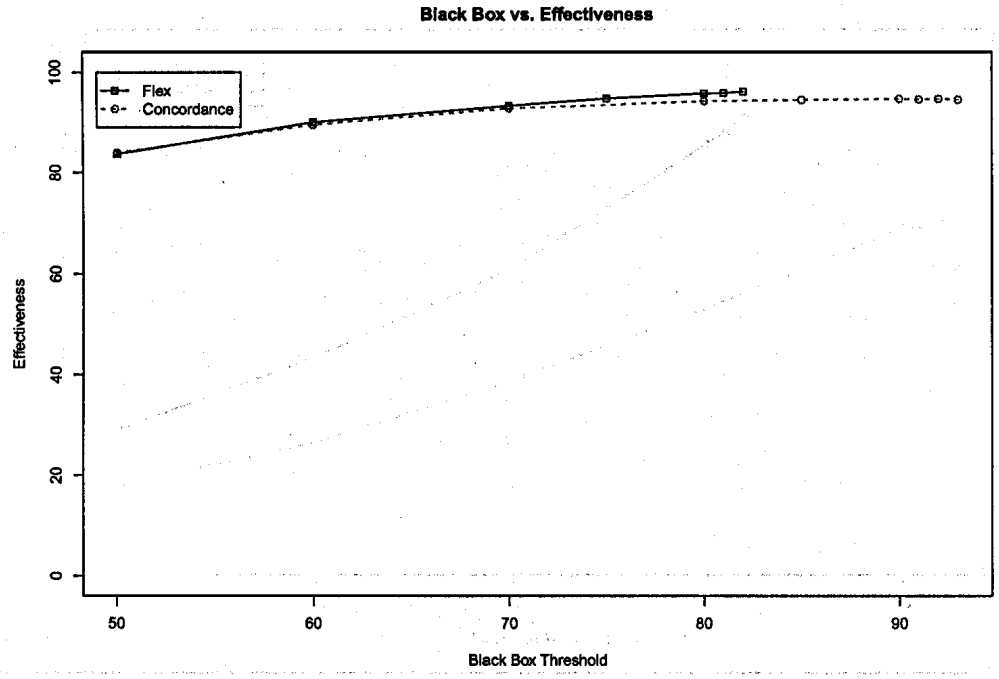


Figure 4.15 The relationship between black box and white box coverage for discrete black box data set

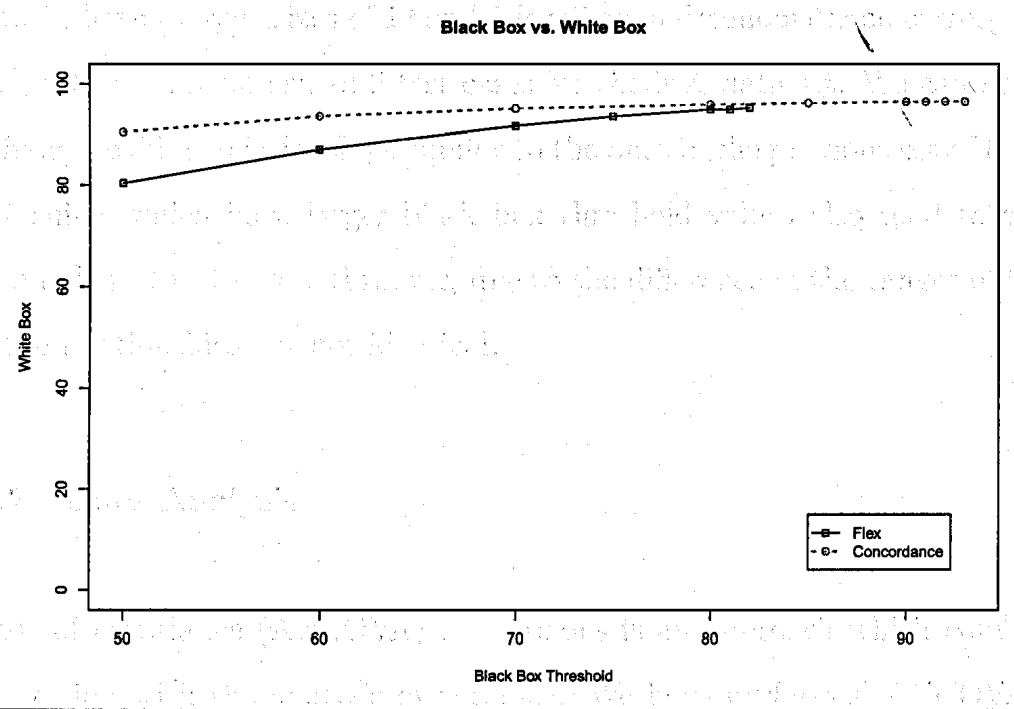
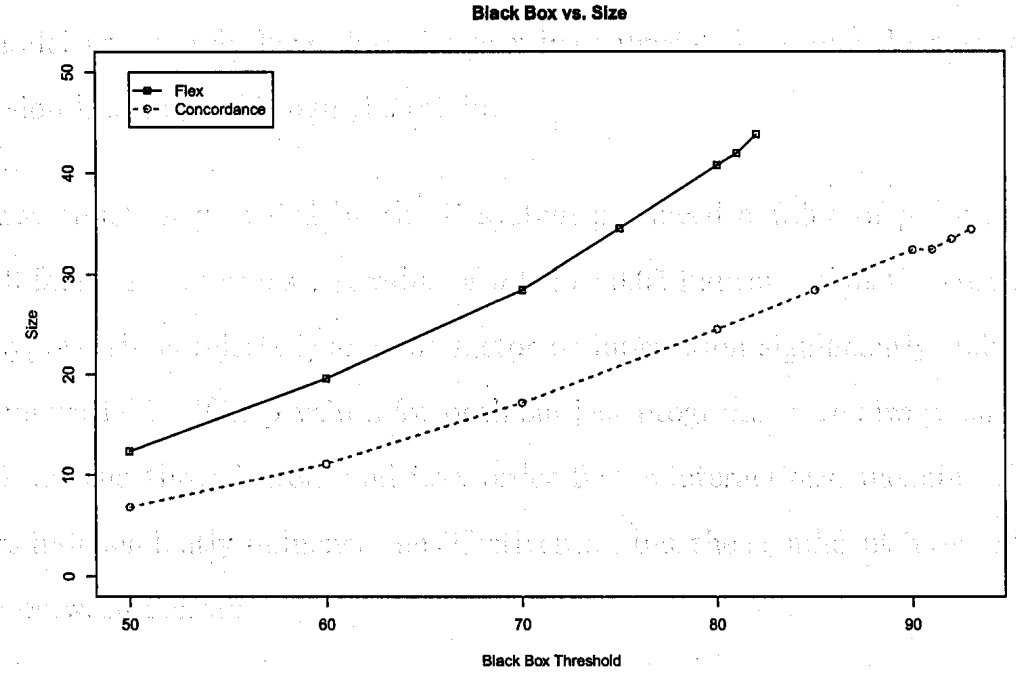


Figure 4.16 The relationship between black box coverage and size for discrete black box data set



figures, each point represents the average value for 1000 test suites with the given black box threshold value. The test suites with the smallest black box threshold value of 50 have average sizes of 12 and 7 for `flex` and `concordance` correspondingly, compared to the lowest size of 2 test cases for the first data set. We can notice that these figures exhibit relationships similar to the ones in the previous set of test suites, as test suites which have larger black box threshold values also tend to contain a larger number of test cases. However, due to the difference in the ranges of test suite sizes, the relationships are not identical.

4.5.3.2 Data Analysis

Analysis of covariance (ANCOVA) in statistics is an approach which combines linear regression with the analysis of variance. We have performed ANCOVA on the

experimental data using the linear model $Eff = B0 + B1*log(size) + B2*black\_box + B3*white\_box$. We include the constant term $B0$ (also called an intercept) into the model as we only have data for positive values of size, and do not expect the regression line to go through the origin.

The aov function provided by the R system produced a value of p for each factor and all factor interactions[1]. A value of p below 0.05 indicates that the corresponding null hypothesis is rejected, and the factor or interaction significantly influences the outcome variable. The p values for both subject programs were always smaller than 0.0001 for the three factors and first order factor interactions, meaning that these factors independently influence the effectiveness, but the significant interaction effect exists between factors.

After ensuring that all factors significantly influence the outcome variable, we perform regression analysis on the data in order to determine the relationship among variables. We use R's linear regression function to fit various linear models to the experimental data. First, we take the model suggested by Andrews in [25] and add the black box coverage factor to it. We then compare this model to 12 other models which we consider to be less accurate. These models consist of the combination of factors size, log(size), black_box, log(black_box) and white_box. We use the adjusted R^2 value reported by the R's `lm` function as a measurement of how well the model fits the data[2]. Adjusted R^2 is a modification of the R^2 value which adjusts for the number of variables used in the model.

Table 4.4 shows adjusted R^2 values for two sets of test suites for all 13 models and two

[1]The R's aov function works as a wrapper to the `lm` function (see below) to perform an analysis of covariance by fitting an analysis of variance model for each value of discrete factor.

[2]The R's `lm` function is used to fit linear models.

Table 4.4 Goodness of fit of models of effectiveness, measured by adjusted R^2

Model of Effectiveness	Adjusted R^2			
	Discrete size data set		Discrete black box data set	
	`flex`	`concordance`	`flex`	`concordance`
size	0.5375	0.4259	0.6706	0.5355
log(size)	0.7151	0.6009	0.7276	0.6187
black_box	0.7001	0.6977	0.6992	0.5947
log(black_box)	0.7437	0.7767	0.7161	0.631
white_box	0.9081	0.8395	0.8942	0.5663
black_box + white_box	0.9085	0.9098	0.8953	0.6261
log(size) + black_box	0.7819	0.7002	0.7295	0.6189
log(size) + white_box	0.9133	0.9079	0.8944	0.635
log(size) + b.b. + w.b	0.9149	0.912	0.9002	0.6353
log(size) + log(b.b.) + w.b	0.9136	0.9119	0.899	0.6441
size + black_box	0.7258	0.7154	0.7	0.6175
size + white_box	0.9087	0.8946	0.8943	0.604
size + b.b. + w.b	0.9096	0.9104	0.8967	0.6408

subject programs. We first analyze the data set of test suites with fixed sizes. The adjusted R^2 values indicate that of all three factors the white box factor influences the effectiveness most of all, while size and black box measures add a little more accuracy to the model. The table shows that the model of the form *Eff = B0 + B1*log(size) + B2*white_box + B3*black_box* has the largest adjusted R^2 value for both subject programs and therefore is considered to be the most accurate model. We also notice that the model of the form *Eff = B0 + B1*log(black_box)* is more accurate than the model of the form *Eff = B0 + B1*black_box*, which is consistent with the shape of the scatterplots in Figures 4.12 and 4.13.

In order to determine if a model which uses logarithmic size is more accurate than the model which uses raw size, we compare various models with linear and logarithmic size factors. We find that models of the form *Eff = B0 + B1*log(size) + B2*white_box* and *Eff = B0 + B1*log(size) + B2*white_box + B3*black_box* are more accurate than

Table 4.5 Coefficients for the linear model *Eff = B0 + B1*log(size) + B2*black_box + B3*white_box.*

Subject	Coefficient			
	B0	**B1**	**B2**	**B3**
`flex`	-5.01	1.61	-0.01	1.11
`concordance`	-0.31	1.637	0.171	0.788

the similar models with linear size. This observation is consistent with the findings of Andrews in [25]. However, the model of the form *Eff = B0 + B1*log(size) + B2*black_box* is slightly less accurate that the corresponding model with linear size. We can also see that *log(size)* and *black_box* add similar amount of accuracy to the model, because the models of the form *Eff = B0 + B1*log(size) + B2*white_box* and *Eff = B0 + B1*black_box + B2*white_box* have adjusted R^2 values which are close to each other. We can conclude from these observations that the *log(size)* and *black_box* factors have similar effect on the outcome variable, and are highly correlated.

The output of R's **anova** function indicates that there exists a statistically significant difference between the model *Eff = B0 + B1*log(size) + B2*white_box + B3*black_box* and all other models, making it the best fitted function for the collected data (p value of t test is less than 0.0001 for all pairs of models). Table 4.5 shows coefficients for this model for the two subject programs. Figures 4.18 and 4.17 show graphs of actual (Y-axis) vs. predicted (X-axis) values for the effectiveness model.

We have performed a similar analysis for the second set of test suites with fixed black box threshold values. The adjusted R^2 values for this data set are shown in Figure 4.4. The adjusted R^2 values for the `flex` subject program are consistent with the values for the first set of data: the white box factor influences the effectiveness most of all; models with logarithmic size are more accurate than the models with linear size;

Figure 4.17 Predicted vs. actual Effectiveness for `concordance` using the model *Eff = B0 + B1*log(size) + B2*black_box + B3*white_box*

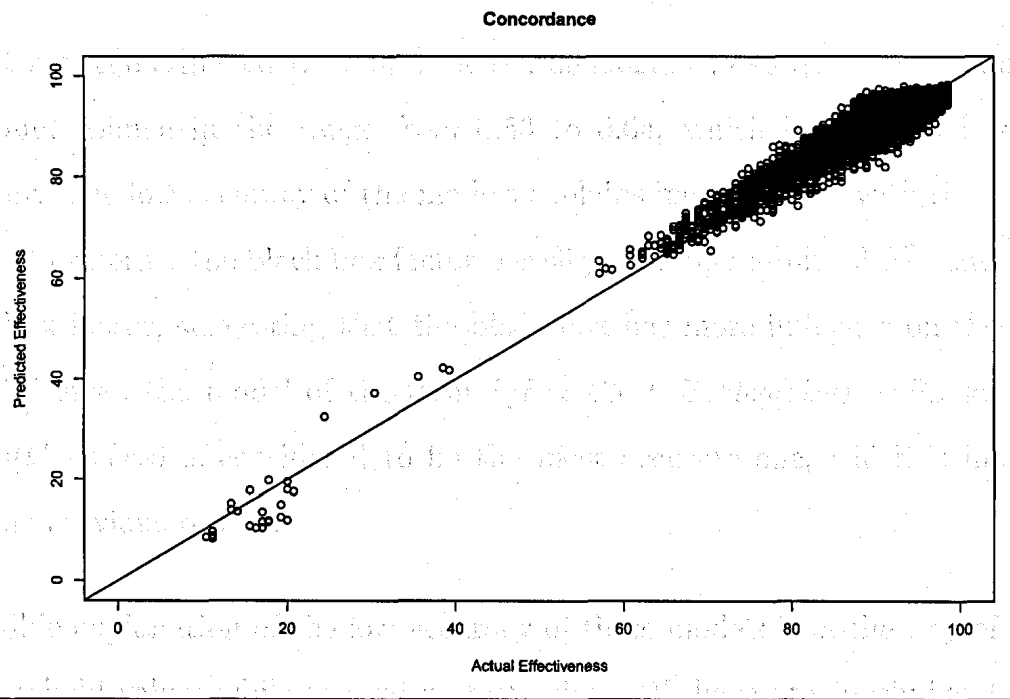
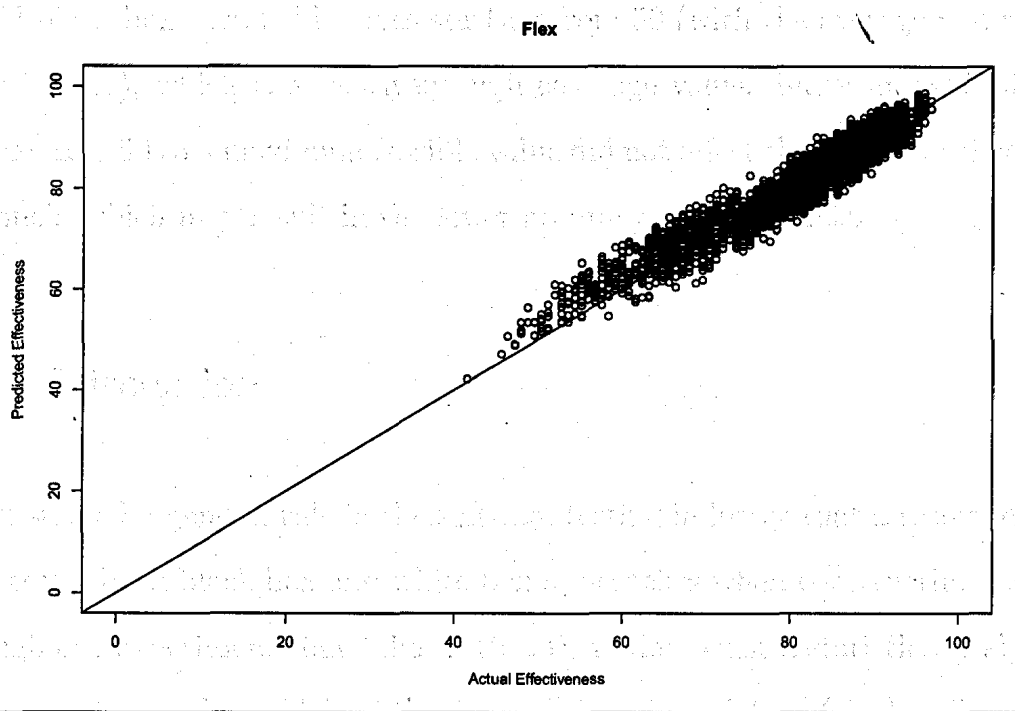


Figure 4.18 Predicted vs. actual Effectiveness for `flex` using the model *Eff = B0 + B1*log(size) + B2*black_box + B3*white_box*



and the model of the form *Eff = B0 + B1*log(size) + B2*white_box + B3*black_box* is considered to be the most accurate one.

In contrast, the adjusted R^2 values for the concordance subject program are much lower and change in the range from 0.53 to 0.64, which is considered to be low accuracy. The low accuracy of the models explains inconsistency with the previously observed patterns: the black box factor has slightly larger adjusted R^2 value than the white box factor, suggesting that the black box has more influence on the outcome variable. Also, the model of the form *Eff = B0 + B1*log(size) + B2*white_box + B3*log(black_box)* is considered to be the most accurate one, which is inconsistent with the previous results.

A possible explanation of the low accuracy of these models is in the way of choosing the threshold values while generating test suites. We have previously identified that the relationship between the black box coverage and effectiveness is logarithmic, and test cases with low black box coverage can ensure high levels of effectiveness. We have selected black box threshold values starting from 50 (with the average test suite size of 8 test cases), which is a relatively high coverage value. Increasing the black box coverage from 50 to a maximum feasible value did not affect the test suite effectiveness very much, which may result in the lower accuracy of the models.

4.5.4 Discussion

It is considered a general rule in the software testing industry that a tester must take into account both black box and white box approaches when constructing test cases. Although our experiments have shown that there does exist a statistically significant difference between the models of the forms *Eff = B0 + B1*log(size) + B2*white_box*

and $Eff = B0 + B1*log(size) + B2*black\_box + B3*white\_box$, the effect of adding the black box factor to the model is relatively small (less than 0.01 difference in adjusted R^2). This difference does not have a large impact in practice. We have also found that the white box coverage alone is better at predicting the effectiveness than the black box coverage alone, while log(black_box) is better at predicting the effectiveness than the raw black box coverage value when size is not taken into account. The logarithmic relationship between the black box coverage and effectiveness means that a test case with low black box coverage is likely to be more effective than a test case with low white box coverage. However when we increase the black box coverage, after a certain point the effectiveness does not increase very much.

Based on these findings, we suggest that a good testing strategy is first to use the black box coverage approach to construct test cases, as this will ensure a high level of effectiveness. After that it is useful to add the white box test cases to the test suite to make sure that the high code coverage is achieved which in turn will lead to the increased effectiveness. This property makes the black box approach suitable for smoke tests and regression tests, in which a small amount of test cases must provide a high level of confidence about the quality of the program.

We also suggest that the black box approach is more suitable at the early stages of software development, before the final release of the product. A tester will use test cases constructed using the black box approach to ensure that no functionality is missing and all features are working as they are designed to. However, when all functionality is implemented, it is more appropriate to run test cases which aim to achieve high code coverage in order to find more bugs. Therefore both black box and white box approaches are equally important, but should be used for different purposes.

Chapter 5

Conclusion

5.1 Conclusion

This thesis has presented a method to evaluate the thoroughness of a test suite from the black box perspective. This method is based on the two most widely used industry black box techniques: equivalence partitioning and boundary value analysis. The method consists of three main components: Functional Test Specification, which defines equivalence classes for each input and output variable; log files, which are produced during the execution of a subject program, and the FTS Tool, which compares elements from log files with elements from the FTS, and calculates the percentage of elements covered. This thesis also presented the architecture and implementation of the FTS Tool coverage calculation program. The design of the tool is scalable and will allow easy addition of other coverage types to the report. Several experiments have been conducted in order to evaluate the proposed method, as well as to study the relationship among the black box and white box coverage measures, test suite

size and effectiveness, and see if the black box coverage can be a good predictor of test suite effectiveness.

We have successfully applied the black box coverage calculation method to three subject programs of different sizes implemented in different programming languages. Experimental results indicate that the white box coverage is better at predicting the effectiveness than the black box coverage. We suggest that the lower accuracy of the black box factor in comparison with the white box factor can be due to the smaller granularity of the black box specification - the number of black box coverage elements is very small compared to the number of lines of code in the program. We have also found that there exists an exponential relationship between the black box coverage and effectiveness, which means that a test case with low black box coverage is likely to be more effective than the test case with low white box coverage. However, when we further increase the black box coverage the effectiveness increases only slightly. We have also found that the model of the form log(size) + black_box + white_box is the most accurate model based on the statistical adjusted R^2 parameter; however, adding the black box coverage factor to the model of the form log(size) + white_box does not significantly increase the accuracy of the model.

Our findings suggest that when constructing test cases, the black box testing approach should be given the preference at the early stages, as the relatively low black box coverage can provide high test suite effectiveness. After achieving a high level of the specification coverage, it is appropriate to add the white box test cases to make sure that high code coverage is achieved. This is consistent with the general testing practices.

5.2 Future Work

Several improvements can be made to the coverage calculation program and experiments. The current implementation of the FTS Tool does not take into account combinations of values of different input variables. For example, if we want to test a system which takes as input a date in the form of a day number and a month number, then valid values for the month variable will be from 1 to 12, and valid values for the day variable will be from 1 to 31. However, certain combinations of these valid values, such as 2/30, 4/31 etc. are invalid. We would like to extend the XML specification definition so that it will allow to specify combinations of value sets and indicate whether they are valid or invalid, and also change the FTS Tool implementation so that it will check if a certain combination of value sets was present in the logs.

One limitation of the current approach has been uncovered during the experiments with the `concordance` subject program. `concordance` has two modes of operation: help printing mode, when the `--help` option is specified, and a regular mode in which an "inputFileName" variable is required (which is captured in the FTS as an invalid multiplicity of "0"). Both of these arguments belong to the input event "OptionsSpecified". So, when the following input event appears in the log file: "InputEvent program commandLine OptionsSpecified Help=–help", the multiplicity of the inputFileName variable is considered to be invalid by the FTS Tool. According to the value set coverage calculation algorithm, a value set can be considered tested if the corresponding input event does not contain invalid variable values or multiplicities. Therefore in this input event the value set "Help=–help" is not considered to be covered. We would like to incorporate into the FTS a way to specify the restriction that certain input variables cannot be used together in one input event, which in turn will be taken into account by the FTS Tool while calculating the coverage.

We also would like to incorporate the following improvements to the experiments:

- Add more test cases to the existing pool of test cases for the two subject programs, so that the test pool will achieve 100% white box and 100% black box coverage. In this case we will not need to scale the coverage values for the experiments, which will potentially improve the accuracy of the linear regression models of effectiveness.

- Test the black box coverage calculation approach on a wider range of subject programs, including programs of larger sizes, programs with GUI and a more diverse functionality. This will allow us to have a more advanced FTS specification, which will increase the number of coverage elements and improve the accuracy of statistical analysis.

- Generate mutants for the `yamm` subject program and perform analysis similar to the one described in Section 4.5.

- In Section 4.5.4 we suggest that the white box technique is more useful for a complete program which does not change over time, while the black box approach is useful during the implementation of the program. In order to test this hypothesis we would like to perform analysis of several different versions of the same program, containing different sets of features.

References

[1] Abbot framework for automated testing of Java GUI components and programs. `http://abbot.sourceforge.net/doc/overview.shtml/`. [Online. Accessed July 2011].

[2] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard*, UML'00, pages 383–395, Berlin, Heidelberg, 2000. Springer-Verlag.

[3] Jean-Raymond Abrial, Stephen A Schuman, and Bertrand Meyer. A specification language. In *A. M. Macnaghten and R. M. McKeag, editors, On the Construction of Programs*, pages 343–410, New York, NY, USA, 1980. Cambridge University Press.

[4] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Proceedings of the 7th Annual Conference on Computer Assurance*, COMPASS '92, pages 15–18, 1992.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 2008.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[7] James H. Andrews. CS4472: "Specification, Testing and Quality Assurance" Lecture Notes. `http://www.csd.uwo.ca/courses/CS4472a/`. [Online. Accessed September 2009].

[8] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32:608–624, August 2006.

[9] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 210–218, New York, NY, USA, 1989. ACM.

[10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/REC-xml/`. [Online. Accessed September 2011].

[11] Cobertura Project Homepage. `http://cobertura.sourceforge.net/`. [Online. Accessed July 2011].

[12] Lee Copeland. *A Practitioner's Guide to Software Test Design.* Artech House, Inc., Norwood, MA, USA, 2003.

[13] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[14] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17:900–910, September 1991.

[15] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng.*, 19:774–787, August 1993.

[16] Phyllis G. Frankl and Oleg Iakounenko. Further empirical studies of test effectiveness. *SIGSOFT Softw. Eng. Notes*, 23:153–162, November 1998.

[17] Gcov - a Test Coverage Program in: Using the GNU Compiler Collection (GCC). `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html/`. [Online. Accessed July 2011].

[18] The RAISE Language Group. *The RAISE specification language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[19] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[20] C B Jones. *Systematic software development using VDM.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.

[21] Jtest Project Homepage. `http://www.parasoft.com/jtest/`. [Online. Accessed July 2011].

[22] LOCC - Collaborative Software Development Laboratory. `http://csdl.ics.hawaii.edu/Plone/research/locc/`. [Online. Accessed October 2011].

[23] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for UML activity diagrams. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 2–8, New York, NY, USA, 2006. ACM.

[24] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979.

[25] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[26] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31:676–686, June 1988.

[27] Mahesh Shirole, Amit Suthar, and Rajeev Kumar. Generation of improved test cases from UML state diagram using genetic algorithm. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 125–134, New York, NY, USA, 2011. ACM.

[28] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22, November 1996.

[29] W. N. Venables, D. M. Smith, and The R Development Core Team. An introduction to R. Technical report, R Development Core Team, June 2006.

[30] Y. T. Yu, Eric Y. K. Chan, and P.-L. Poon. On the coverage of program code by specification-based tests. In *Proceedings of the 2009 Ninth International Conference on Quality Software*, QSIC '09, pages 41–50, Washington, DC, USA, 2009. IEEE Computer Society.

[31] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29:366–427, December 1997.