A Test Coverage Analyser for Software Testing

Willem Myburgh
Department of Computer Science
University of Stellenbosch
South Africa

HONOURS PROJECT 1999

Abstract

Software testing is an essential part of the software development cycle. It entails the identification of test cases and the execution of them to see if the program behaves in the expected way. An objective gauge for the quality of a program could be defined by the coverage of these test cases. There exist test coverage metrics to measure the extent to which a set of test cases covers a program.

We implemented a Test Coverage Analyser. A Test coverage analyser is a tool that shows the coverage of the test cases while the tester is running the test cases. This is done by instrumenting the source code under test, before it is being tested. The analyser will colour the source code that was executed to give a visual report of the coverage. The tester will exactly know which code was not executed.

It also will show how well all the predicates in conditional control statements are tested. Every simple predicate in a compound predicate is considered in the testing. The tester will know what outcome, for a specific simple predicate, was not tested.

This document reports on the design and implementation of a prototypical test coverage analyser for the *Native Oberon* (see [3]) system.

Contents

List of Figures

Preface

This document is a report of an honours project that was done in the department of Computer Science at the University of Stellenbosch. The supervisor for the project was PJA de Villiers.

Willem Myburgh
11 February 2000

Chapter 1

Introduction

This honours project investigated an approach that can lead to an improvement of software quality. It will help software testers to apply good testing techniques. We have developed a prototypical test coverage analyser (TCA) in *Native Oberon*. It is a software testing tool to help the tester to test a software item thoroughly.

Every source code example in this report will be based on the *Oberon* language.

Before we describe exactly what a test coverage analyser is, let us look first what software testing is.

1.1 Software Testing

Software needs to be tested after the development phase. We test because we know that we are capable of making mistakes and we want to discover the problems due to our mistakes.

The essence of software testing is to determine a set of *test cases* for the software item being tested. A test case defines the inputs to the software item being tested and the expected output. A software item could be a module, procedure or just a part of a module or procedure. There are two fundamental approaches to identifying test cases: *functional testing* (section 1.1.1) and *structural testing* (section 1.1.2).

1.1.1 Functional Testing

Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range.With this approach to test case identification, the only information that is used is the specification of the software. It is also known as *Black Box Testing* (figure 1.1), because the test cases are independent of the implementation. One of the fundamental limitations of functional testing is

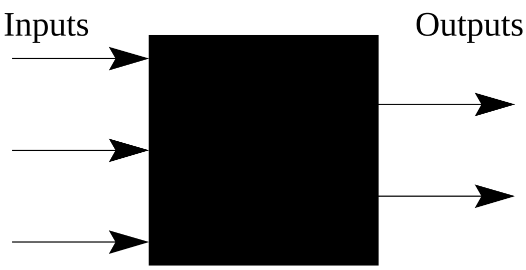

Figure 1.1: A Black Box

that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program.

1.1.2 Structural Testing

Structural testing, sometimes called *White Box Testing*, uses the implementation to identify test cases. The structure of a program can be seen as a graph. With concepts of linear graph theory (section 1.2), the tester can rigorously describe exactly what is being tested. Because of this theoretical basis, structural testing lends itself to the definition and use of *test coverage metrics* (section 2.1). Test coverage metrics provide a way to explicitly state the extent to which a software item has been tested.

If the coverage of test cases is good enough, structural testing could help to find problems at logical expressions in conditional control statements.

Both functional and structural testing are needed: When functional test cases are executed in combination with structural test coverage metrics, both expected behaviour and undocumented behaviour can be tested.

1.1.3 Example

The *Triangle Problem* is a classical problem when it comes to explaining software testing. An implementation is given in figure 1.2. It accepts three integers as input which are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides. This example will be used in other sections of this report.

1.2 Graphs for Testing

The most common use of graph theory in software testing is the *program graph*. The following definition is given by Jorgensen [5]:

Definition: Given a program written in an imperative programming language, its *program graph* is a directed graph in which nodes are either

```
MODULE Test;
  IMPORT In, Out;

  PROCEDURE Triangle*;
    VAR
      a, b, c: INTEGER;
      IsATriangle: BOOLEAN;
  BEGIN
    In.Open;
    In.Int(a); In.Int(b); In.Int(c);

    IF (a < b + c) & (b < a + c) & (c < a + b) THEN
      IsATriangle := TRUE
    ELSE
      IsATriangle := FALSE
    END;

    IF IsATriangle THEN
      IF (a = b) & (b = c) THEN
        Out.String("Triangle is Equilateral"); Out.Ln
      ELSIF (a # b) & (a # c) & (b # c) THEN
        Out.String("Triangle is Scalene"); Out.Ln
      ELSE
        Out.String("Triangle is Isosceles"); Out.Ln
      END
    ELSE
      Out.String("Not a Triangle"); Out.Ln
    END
  END Triangle;
END Test.
```

Figure 1.2: An implementation of the Triangle Problem

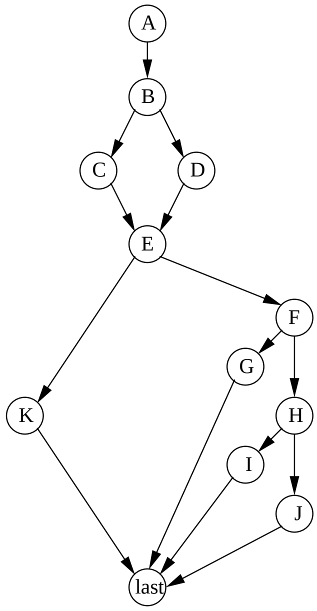

Figure 1.3: DD-path graph for the Triangle Problem

entire statements or fragments of a statement, and edges represent flow of control.

If i and j are nodes in the program graph, there is an edge from node i to node j if and only if the statement (fragment) corresponding to node j can be executed immediately after the statement (fragment) corresponding to node i.

A sequence of statements where there are no internal branches can be group together. This sequence is known as a decision-to-decision path (DD-path). Jorgensen [5] defined DD-paths in terms of a chain in a program graph. A *chain* is a path in which the first node and last node are distinct, and the in-degrees and out-degrees of the internal nodes are one.

Definition: A *DD-path* is a chain in a program graph such that it consists of a single node with in-degree $= 0$, or out-degree $= 0$, or in-degree ≥ 2, or out-degree ≥ 2, or it consists of a single node with in-degree $= 1$ and out-degree $= 1$, or it is a maximal chain of length ≥ 1.

A *DD-path graph* is then a directed graph in which nodes are DD-paths of their program graph, and edges represent control flow between successor DD-paths. The DD-Path graph of the Triangle problem is shown in figure 1.3.

1.3 Overview

Chapter 2 makes it more clearer what a test coverage analyser do by introducing test coverage metrics. After that a definition is given of a *basic block* and how it is instrumented by the TCA. A discussion on the instrumentation of predicates in conditional control statements is also given.

In chapter 3 the implementation of the TCA is discussed. It gives information on the environment and the compiler.

A brief discussion on the testing of the test coverage analyser is given in chapter 4. It is also evaluated in terms of the sizes and execution times of the resulting instrumented object code.

Chapter 5 will give a summary of the report and also suggestions for future work. It ends with a conclusion.

Appendix A is a user's manual for the test coverage analyser.

Chapter 2

What is a Test Coverage Analyser?

We gave some preliminary definitions in the previous chapter. In this chapter we are discussing more definitions regarding test coverage metrics to make it clear what our test coverage analyser do.

2.1 Test Coverage Metrics

DD-Paths enable very precise descriptions of test coverage. Test coverage metrics are a device to measure the extent to which a set of test cases covers a program and DD-Paths help to define how to test specific metrics. There are several widely accepted test coverage metrics. Table 2.1 gives a list of test coverage metrics. Some are described in the following subsections.

2.1.1 Statement Coverage

This coverage metric requires that a set of test cases is found such that, when executed, every node of the program graph is traversed at least once.

Metric	Description of Coverage
C_0	Every statement
C_1	Every DD-Path (predicate outcome)
C_1p	Every predicate to each outcome
C_{MCC}	Multiple condition coverage
C_ik	Every program path that contains up to k repetitions of a loop
C_∞	All possible execution paths

Table 2.1: Structural test coverage metrics

It means that every statement should be executed. This could also be defined as basic block coverage (see section 2.3.1).

2.1.2 DD-Path Testing / Branch Coverage

If every edge in the program graph is traversed, then every DD-Path is covered by the test cases. This means that, for an If-statement, both the *true* and *false* branches are covered. This coverage metric includes statement coverage.

2.1.3 Multiple Condition Coverage (MCC)

In conditional control statements every simple predicate in a compound predicate must be tested for both their *true* and *false* outcomes. That is all the different ways that each outcome can occur, must be investigated. For example, consider the following case:

```
IF X & Y THEN
  Out.String("Hello"); Out.Ln
END
```

MCC requires three cases:

- X is *true*, Y is *true*
- X is *true*, Y is *false*
- X is *false*

Note that *Oberon* uses short-circuit evaluation. Short-circuit evaluation is defined for the two boolean operators as follows:

p OR q $\Rightarrow$ if p then TRUE else q
p & q $\Rightarrow$ if p then q else FALSE

The MCC metric includes the branch coverage metric discussed in section 2.1.2.

2.2 Test Coverage Analysers

A test coverage analyser offers automated support for the coverage metrics approach to test management. With a coverage analyser, the tester runs a set of test cases on a program that has been "instrumented" by the coverage analyser. The analyser then uses information produced by the instrumentation code to generate a coverage report or a coverage map. In this way, the tester can experiment with different sets of test cases to determine the coverage of each set. The components in a test coverage analyser is shown in figure 2.1.

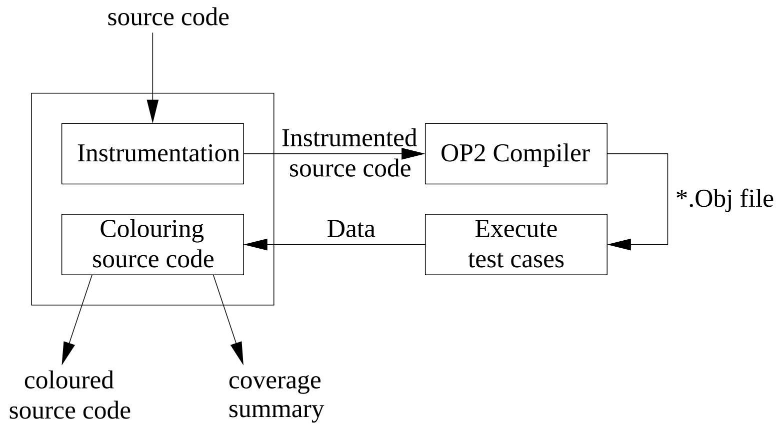

Figure 2.1: The components and data flow in a test coverage analyser

2.3 Instrumentation

When a program item needs to be tested, the source code of that item's module must be given as input to the test coverage analyser. The TCA then will add extra source code to the given source code and compile the resulting source code. This extra code are, in our implementation, counters, the procedures to initialize them and to report their values after execution. A specific counter is associated with a specific component in the source code. The following two subsections will describe how the two most important components in the source code are instrumented by these counters.

2.3.1 Basic Blocks

The first coverage metric that was discussed in this report was the statement coverage metric (section 2.1.1). To instrument the source code for testing this metric, the *basic blocks* in the source code must be identified. Aho *et al* [1] defines a *basic block* as follows:

Definition: A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

This definition must be used, for instrumentation purposes, at the programming language level and not at a lower level. All statements in a *basic block* are executed as one unit. Thus to know whether a specific statement was executed, it is only necessary to instrument the *basic block* where that statement is situated. The *basic blocks* in the Triangle problem are shown in figure 2.2. Note that all control statements are excluded from a *basic block*. The instrumentation of the *basic blocks* is shown in figure 2.3.

The only statements in *Oberon* that can be in a *basic block*, are

```
MODULE Test;
  IMPORT In, Out;

  PROCEDURE Triangle*;
    VAR
      a, b, c: INTEGER;
      IsATriangle: BOOLEAN;
  BEGIN
    In.Open;
    In.Int(a); In.Int(b); In.Int(c);

    IF (a < b + c) & (b < a + c) & (c < a + b) THEN
      IsATriangle := TRUE
    ELSE
      IsATriangle := FALSE
    END;

    IF IsATriangle THEN
      IF (a = b) & (b = c) THEN
        Out.String("Triangle is Equilateral"); Out.Ln
      ELSIF (a # b) & (a # c) & (b # c) THEN
        Out.String("Triangle is Scalene"); Out.Ln
      ELSE
        Out.String("Triangle is Isosceles"); Out.Ln
      END
    ELSE
      Out.String("Not a Triangle"); Out.Ln
    END
  END Triangle;
END Test.
```

Figure 2.2: The *basic blocks* in the Triangle Problem

```
MODULE Test;
  IMPORT In, Out;
  (* Procedures and declarations for instrumentation *)

  PROCEDURE Triangle*;
    VAR
      a, b, c: INTEGER;
      IsATriangle: BOOLEAN;
  BEGIN
    INC(Counters[0]);
    In.Open;
    In.Int(a); In.Int(b); In.Int(c);

    IF (a < b + c) & (b < a + c) & (c < a + b) THEN
      INC(Counters[1]);
      IsATriangle := TRUE
    ELSE
      INC(Counters[2]);
      IsATriangle := FALSE
    END;

    IF IsATriangle THEN
      IF (a = b) & (b = c) THEN
        INC(Counters[3]);
        Out.String("Triangle is Equilateral"); Out.Ln
      ELSIF (a # b) & (a # c) & (b # c) THEN
        INC(Counters[4]);
        Out.String("Triangle is Scalene"); Out.Ln
      ELSE
        INC(Counters[5]);
        Out.String("Triangle is Isosceles"); Out.Ln
      END
    ELSE
      INC(Counters[6]);
      Out.String("Not a Triangle"); Out.Ln
    END
  END Triangle;
BEGIN
  (* Initialization of Counters *)
END Test.
```

Figure 2.3: The instrumentation of the *basic blocks* in the Triangle Problem

- Assignments,
- Procedure calls, except for the *HALT* procedure that can only be called at the end,
- the *EXIT* statement, only at the end, and
- the *RETURN* statement, only at the end.

The instrumentation on the predicates in conditional control statements will be discussed in section 2.3.2.

2.3.2 Conditional Control Statements

To determine multiple conditional coverage (section 2.1.3), all the simple predicates in a compound predicate must be isolated to determine their individual outcomes. There can be a few approaches for the instrumentation. The two main approaches for instrumentation are the following:

- Leave the original boolean expression as it is and determine after the evaluation the appropriate outcomes for the simple predicates, or
- Replace the entire boolean expression by a nested simple *IF-THEN-ELSE* logic that simulates the short-circuit evaluation to determine the outcomes for the simple predicates.

An example of both approaches is shown in figures 2.5 and 2.6. They both instrument the source code in figure 2.4. The outcomes for the simple predicates A, B and C for the two approaches are determined, respectively, as follows:

- A was *true* if Counters[1] or Counters[3] is greater than zero.
  A was *false* if Counters[2], Counters[4] or Counters[5] is greater than zero.
  B was *true* if Counters[2] or Counters[4] is greater than zero.
  B was *false* if Counters[5] is greater than zero.
  C was *true* if Counters[1] or Counters[2] is greater than zero.
  C was *false* if Counters[3], Counters[4] or Counters[5] is greater than zero.
- A was *true* if Counters[1] is greater than zero.
  A was *false* if Counters[2] is greater than zero.
  B was *true* if Counters[3] is greater than zero.
  B was *false* if Counters[4] is greater than zero.
  C was *true* if Counters[5] is greater than zero.
  C was *false* if Counters[6] is greater than zero.

```
IF (A OR B) & C THEN
  Out.String("Hello"); Out.Ln
END
```

Figure 2.4: A portion of source code to instrument

```
IF (A OR B) & C THEN
  IF A THEN INC(Counters[1])
  ELSE INC(Counters[2]) END;
  Out.String("Hello"); Out.Ln
ELSIF A THEN INC(Counters[3])
ELSIF B THEN INC(Counters[4])
ELSE INC(Counters[5])
END
```

Figure 2.5: Instrumentation after evaluation

```
IsTRUE := TRUE;
IF A THEN INC(Counters[1]) ELSE INC(Counters[2]);
  IsTRUE := FALSE
END;
IF ~IsTRUE THEN
  IF B THEN INC(Counters[3]); IsTRUE := TRUE
  ELSE INC(Counters[4])
  END
END;
IF IsTRUE THEN
  IF C THEN INC(Counters[5]) ELSE INC(Counters[6]);
    IsTRUE := FALSE
  END
END;
IF IsTRUE THEN
  Out.String("Hello"); Out.Ln
END
```

Figure 2.6: Replacement of entire boolean expression by instrumentation

The first approach has two important drawbacks: Firstly the simple predicates are evaluated twice and secondly it is a complex task to determine the formulation of the instrumentation when the compound predicate becomes large. When a simple predicate is evaluated twice and it consists of a function call, side effects can put the program into an inconsistent state. This could be corrected by declaring a variable to store the returned value of the function at first and then reference that variable in the evaluations, replacing the function call.

One of the biggest drawbacks of the second approach is that the instrumentation leads to much more code to be generated. Another drawback is that the compiled code of the original source is changed and this could prevent the detection of a bug in the compiler. The advantage is that the instrumentation follows the short-circuit evaluation logic to evaluate an expression and at the same time it knows what the outcomes of each simple predicate are. This approach seems to be the more effective one and is the one we have decided to use. The algorithm to generate the instrumentation for it will be explained in section 3.4.3.

```
IF B1 THEN S1
ELSIF B2 THEN S2
...
ELSIF Bn THEN Sn
ELSE S
END
```

```
Evaluate B1;
IF IsTRUE THEN S1
ELSE Evaluate B2;
  IF IsTRUE THEN S2
  ELSE
    ...
    ELSE Evaluate Bn;
      IF IsTRUE THEN Sn
      ELSE S
      END
    END
  END
END
```

Figure 2.7: Instrumenting an If statement

```
WHILE B DO
S
END
```

```
Evaluate B;
WHILE IsTRUE DO
  S;
  Evaluate B
END
```

Figure 2.8: Instrumenting a While statement

Figures 2.7, 2.8 and 2.9 shows how the If statement, While statement and Repeat statement, respectively, are instrumented with regard to predicate evaluation.

2.4 Coverage Report

After the execution of the test cases, the test coverage analyser can give a coverage report. This is done by colouring the source code of the item under test. Our test coverage analyser uses the following colour scheme:

- Green: Code that was executed and predicates that was tested for *true* and for *false*.
- Red: Code that was not executed.
- Blue: Predicates that was tested for *true* only.
- Purple: Predicates that was tested for *false* only.

```
REPEAT
  S
UNTIL B
```

```
REPEAT
  S;
  Evaluate B
UNTIL IsTRUE
```

Figure 2.9: Instrumenting a Repeat statement

With this colouring the tester can see how the code was tested up to the multiple condition coverage metric.

In addition to the coverage, a summary is also given in the *Oberon Log*. The summary consists of a percentage of basic blocks executed and a percentage of predicates that was tested for both outcomes.

Chapter 3

Implementation

In the previous chapter we saw what a test coverage analyser do. In this chapter we are discussing the implementation of the TCA.

3.1 Native Oberon Environment

The original Oberon operating system was designed and implemented by Niklaus Wirth and Jürg Gutknecht [12] and a later version was ported to the Intel PC which was called Native Oberon (see [3]).

In Oberon a module is loaded in memory when a procedure in that module has to be executed and stays in memory until it is explicitly unloaded by the user. This concept is an advantage for testing, because after the execution of a command the state of the module remains in memory. The data that was generated by the instrumentation code in a specific module can then be accessed to create a coverage report.

We have decided that every module that is instrumented must be the host of its coverage data and the command to generate a coverage report. The drawback is that after the instrumentation of a module, its interface has changed which means that every module that imports the instrumented module must be recompiled. As an improvement, the coverage data of all the modules under test could be managed by one dedicated module, but then it must be remembered to remove or reset the data of a specific module when that particular module is unloaded or reloaded by an user action.

3.2 Oberon Compiler (OP2)

The *OP2 compiler* (see [2]) is the latest compiler for the Oberon language and the Oberon-2 language (see [11]). It is a portable compiler and we used the front-end for the test coverage analyser. The front-end does the lexical and syntactic analysis. It builds a symbol table and an abstract syntax tree.

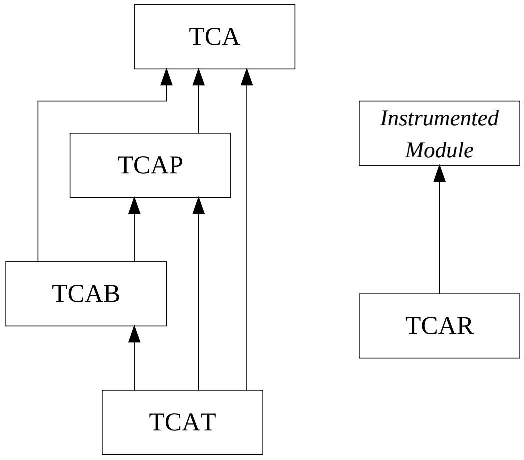

Figure 3.1: The module structure of the test coverage analyser

The OP2 compiler restricts the code size of a module to 65 000 bytes (defined in OPO.Mod) and it could happen that, after the instrumentation of a whole module, the compiler gives a *code too long* error message due to this restriction. In that case the only way is to instrument just a part of the module. For a user's manual see appendix A.

3.3 Module Structure

The module structure of the test coverage analyser is shown in figure 3.1. The topmost module (TCA) is the interface to the user to instrument a module or part of a module. The TCAP module is the original OPP module of the OP2 compiler that was adapted to build a data structure that is used to instrument the given source code. This structure will be discussed in section 3.4.1. The TCAT module is the OPT module that was changed to store the begin and end text positions of the different components in the source code that must be instrumented. The other module TCAB is the syntax tree builder. These modules also import other modules from the OP2-compiler: Compiler, OPM and OPS.

The TCAR module contains the routines that an instrumented module calls to generate a coverage map or report (the colouring of the source code).

3.4 Instrumentation

The instrumentation process consists of three main tasks:

- Collecting information on the different components that need to be instrumented.

```
CompDesc = RECORD
  blockno: INTEGER;
  class: SHORTINT;
  beg, end: LONGINT;
  expr: TCAT.Node;
  next: Component
END;
```

Figure 3.2: The CompDesc record

- Adding the actual instrumentation at the appropriate positions according to the above mentioned information
- Storing the information that is needed to generate a coverage map.

These tasks will be discussed in the following subsections.

3.4.1 Collecting Information

The different components that is need for instrumentation is the following:

- Basic blocks as defined in section 2.3.1,
- IF-*booleanExpression*-THEN fragments,
- ELSIF-*booleanExpression*-THEN fragments,
- END of an If statement,
- UNTIL-*booleanExpression* fragments,
- WHILE-*booleanExpression* DO fragments and
- END of a While statement.

We have decided to store the components in a link list in the order of appearance in the source code. This list is traversed once while the instrumentation code is inserted. A Component is a pointer to record called CompDesc. Its declaration is shown in figure 3.2. A block in the Oberon language can be defined as the declarations and statement sequence between a procedure or module heading and its end. The parser assigns a unique number to each block. The blockno field holds the block number in which the component occur. The class field determines the different components that is listed above. The begin and end text positions of a component is stored in beg and end respectively.

If the component class contains a boolean expression, the pointer to the root node of the expression's parse tree is stored in expr. It is used to break the expression into simple predicates and to determine the evaluation logic of the expression. This will be discussed in the next section. The next field points to the next component in the linked list.

The above mentioned data is collected while parsing statement sequences and expressions.

3.4.2 Add Instrumentation to Code

After the component linked list is built, it must be traversed to add source code to the existing source code. The resulting source code is then saved in file with a name that is the module name together with an extension *.tca.Mod*. The following paragraphs discuss the code that is added to the original source code.

At the end of the import list two modules are added, namely Texts and TCAR. Constants that are added to the source code are

- the number of counters in the instrumented source code (NumOfCouNterS).
- the name of the module to be tested (ModUleNAmE). It is used to verify the module name when a report is generated.
- the name of the file that stores all the text positions of the different components that must be coloured when a report is requested. (PosItioNsFilE). The layout of this file is discussed in section 3.4.4.

The reason for the unconventional use of uppercase letters is to prevent multiple definitions of an identifier that could be defined in the original source code.

The counters to determine what code and how expressions were executed, are defined as

```
cOuNterS: POINTER TO ARRAY NumOfCouNterS OF LONGINT;
```

The boolean iStRuE is used to evaluate the predicates in conditional control statements as shown in section 2.3.2 with IsTRUE.

The InItTeStinG procedure initializes the elements in the array of counters to zero. The exported command to colour the module source code, is declared as ReportTest. It calls the Colour procedure in the TCAR module.

At the beginning of the module's statement sequence, the counters are initialized.

Both the existence of an import list and a statement sequence for the module to be instrumented must be determined to decide whether to include the IMPORT and BEGIN keywords at their respective positions in the resulting source text.

3.4.3 Instrumentation Code to Evaluate Predicates

This section will describe the procedure used to generate the instrumentation code for the evaluation of predicates in conditional control statements. The algorithm traverses the parse tree of the boolean expression in postorder. It is shown in figure 3.3. The truthVal parameter is used to minimize the assignments to the iStRuE boolean. It holds the current truth value of iStRuE in the context where the instrumentation is taking place.

If the predicate in an If or While statement is a simple predicate, the Evaluate procedure is not called because no further break up into simple predicates is needed. However, the *false* branch of the simple predicate needs to be instrumented. This is done by adding an *ELSE* clause to the If statement if there is not one or to add a counter after the While statement. The *UNTIL* clause cannot be optimized in this manner, because there is no other way to determine if the simple predicate was tested for the *false* outcome but to evaluate it before the *UNTIL* clause as shown in figure 2.9.

Let us look at an example boolean expression:

$$\mathsf{((b >= 3)\ \&\ (b < 10))\ OR\ \sim((s = 5)\ \&\ (c <= 6)\ \&\ (b = 2))}$$

The parse tree for this expression is given in figure 3.4. The evaluation of this boolean expression is given in figure 3.5. The numbered lines correspond to the numbered boolean operator nodes.

3.4.4 The Positions File

The positions file is created while the instrumentation is taken place. It associates the text positions and the index's of the counters array to basic blocks or simple expressions. This file is read when a coverage report is generated. The structure of the positions file is defined by the following syntax. ExpTag = 1 and BBTag = 2 are two terminal symbols to distinguish between the two elements and take one byte. The other terminals' sizes are specified by ": len".

```
PositionsFile = {element}
element       = exprBlock | basicBlock
exprBlock     = ExpTag begPos:4 endPos:4
                nofTrueCount:2 {trueCounterIdx:4}
                nofFalseCount:2 {falseCounterIdx:4}
basicBlock    = BBTag begPos:4 endPos:4 counterIdx:4
```

The *begPos* and *endPos* terminals store the the begin and end text positions of the elements in the source code to be coloured. The *nofTrueCount* terminal gives the number of *trueCounterIdx* terminals that follow it. The *trueCounterIdx* terminal stores an index of a counter in the cOuNterS array that is in the true branch of the associated expression evaluation. The

```
PROCEDURE Evaluate(expr: TCAT.Node; truthVal: SHORTINT);
BEGIN (* expr is a boolean expression *)
  IF truthVal = Eunknown THEN
    Add "iStRuE := TRUE;";
    truthVal := Etrue
  END;
  IF expr.class = Nmop THEN
    IF expr.subcl = not THEN
      Evaluate(expr.left, truthVal);
      Add "iStRuE := ~iStRuE;"
    ELSIF expr.subcl = is THEN
      Add "IF" + Is expression + "THEN";
      Add "INC(cOuNterS[" + total + "]);";
      INC(total);
      IF truthVal = Efalse THEN Add "iStRuE := TRUE" END;
      Add "ELSE"
      Add "INC(cOuNterS[" + total + "]);";
      INC(total);
      IF truthVal = Etrue THEN Add "iStRuE := FALSE" END;
      Add "END;"
    END
  ELSIF expr.class = Ndop THEN
    IF expr.subcl = and THEN
      Evaluate(expr.left, truthVal);
      Add "IF iStRuE THEN";
      Evaluate(expr.right, Etrue);
      Add "END;"
    ELSIF expr.subcl = or THEN
      Evaluate(expr.left, truthVal);
      Add "IF ~iStRuE THEN";
      Evaluate(expr.right, Efalse);
      Add "END;"
    ELSE (* Relation *)
      Add "IF" + relational expression + "THEN";
      Add "INC(cOuNterS[" + total + "]);";
      INC(total);
      IF truthVal = Efalse THEN Add "iStRuE := TRUE" END;
      Add "ELSE";
      Add "INC(cOuNterS[" + total + "]);";
      INC(total);
      IF truthVal = Etrue THEN Add "iStRuE := FALSE" END;
      Add "END;"
    END
  ELSE
    (* Designator *)
    Add "IF" + designator expression + "THEN";
    Add "INC(cOuNterS[" + total + "]);";
    INC(total);
    IF truthVal = Efalse THEN Add "iStRuE := TRUE" END;
    Add "ELSE";
    Add "INC(cOuNterS[" + total + "]);";
    INC(total);
    IF truthVal = Etrue THEN Add "iStRuE := FALSE" END;
    Add "END;"
  END
END Evaluate;
```

Figure 3.3: The algorithm for the generation of instrumentation code for predicates in conditional control statements

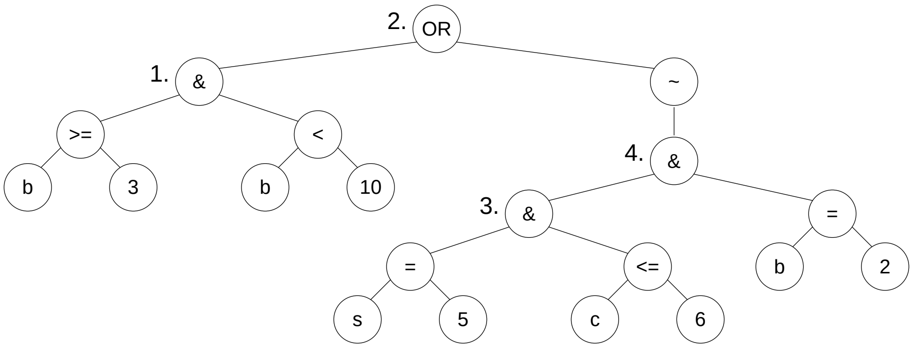

Figure 3.4: The parse tree of a boolean expression

```
    iStRuE := TRUE;
    IF (b >= 3) THEN INC(cOuNterS[0]);
    ELSE INC(cOuNterS[1]); iStRuE := FALSE END;
1.  IF iStRuE THEN
      IF (b < 10) THEN INC(cOuNterS[2]);
      ELSE INC(cOuNterS[3]); iStRuE := FALSE END;
    END;
2.  IF ~iStRuE THEN
      IF (s = 5) THEN INC(cOuNterS[4]); iStRuE := TRUE
      ELSE INC(cOuNterS[5]); END;
3.    IF iStRuE THEN
        IF (c <= 6) THEN INC(cOuNterS[6]);
        ELSE INC(cOuNterS[7]); iStRuE := FALSE END;
      END;
4.    IF iStRuE THEN
        IF (b = 2) THEN INC(cOuNterS[8]);
        ELSE INC(cOuNterS[9]); iStRuE := FALSE END;
      END;
      iStRuE := ~iStRuE;
    END;
```

Figure 3.5: The evaluation of a boolean expression

nofFalseCount terminal gives the number of *falseCounterIdx* terminals that follow it. The *falseCounterIdx* terminal stores an index of a counter in the cOuNterS array that is in the false branch of the associated expression evaluation. In *basicBlock* the *counterIdx* terminal stores the index of the counter that is in the associated basic block.

Chapter 4

Evaluation of the Test Coverage Analyser

This chapter gives an overview of the testing of the TCA as well as an evaluation of it in terms of code size and execution time.

4.1 Testing the TCA

The structural testing methodology that is presented in this paper was used in the testing of the test coverage analyser. Some of the modules of OP2 compiler that were used in the TCA was instrumented and the behaviour of the instrumented modules were compared to the behaviour of the same modules without instrumentation code.

An error that changed the logic of the instrumented code due to the minimization of the assignments to the iStRuE boolean, was detected by making use of the TCA at that stage.

Test modules were written which contained all the different components that could be instrumented.

4.2 Comparison of Instrumented Code with Original Code

When instrumentation code is inserted into source code the resulting source code is bigger and thus results in a larger object file. The performance of the compiled code can also be affected by the instrumentation.

These issues were investigated empirically. We took a wide spectrum of source code and instrument them with the TCA. This is summarized in table 4.1. The *TCAP* module was after the instrumentation too large to be compiled.

Module Name	Number of basic blocks	Number of simple predicates	Original code size	Instrumented code size
Test	9	9	483	1 304
Test2	21	10	713	1 611
Diagrams	23	24	2 081	3 525
TCAT	389	300	17 962	40 537
TCAP	940	806	43 489	>65 000

Table 4.1: Comparisons of the original object code and the instrumented code of modules with different amounts of basic blocks and simple predicates

Original execution time	Instrumented execution time
0.011 sec	0.03 sec
0.064 sec	0.065 sec
0.364 sec	0.519 sec
0.384 sec	0.429 sec
6.429 sec	6.478 sec

Table 4.2: Comparisons of the execution times of original object code and instrumented code

A few comparisons of the execution time between original code and instrumented code is shown in table 4.2. The code was executed and timed for ten times to get an average execution time. It shows that there is a slight degradation in the execution times after instrumentation.

Chapter 5

Summary and Conclusion

5.1 Summary

We have implemented a prototypical test coverage analyser. By instrumenting source code, it uses statement coverage, branch coverage and multiple condition coverage metrics to provide a report on the coverage of test cases. This information helps software developers and testers to focus on the untested areas of code, instead of repeating their efforts on previously tested sections. This helps to improve software quality.

5.2 Future Work

This test coverage analyser that we implemented is just a prototype. It could be enhanced by the following improvements or extensions:

- The instrumentation of some components in the source code can also determine the coverage of other components indirectly so that is not necessary to instrument all the components. Probert [7] discusses techniques for the optimal insertion of instrumentation code. This will help in reducing the compiled code size.

- A network interface can help in testing software on remote machines. This leads to the testing of computer network protocols.

- The test coverage analyser can be integrated with a database system to support software testing. Roussopoulos and Yeh [8] implemented a database system that manage the data required for software testing. For example the positions of components could be stored in such a system. Instrumentation code could communicate values of variables during execution of test cases.

- The instrumentation could be integrated with the compiler so that no instrumented source file is created. The syntax tree could be augmented by instrumentation sub-trees during compilation when the option is set.

- The TCA doesn't use the relation coverage metric. This could also be implemented. Relational coverage requires each possible misuse of relational operators to be covered.
- Support for automatic test case generation will help the tester to identify test cases for untested source code. Kuo-Chung Tai [9] gives a theoretical introduction of test case generation.
- A graphical user interface in *Gadgets* can be added.

5.3 Conclusion

Computer users expect quality software applications because it affects their productivity. Software developers should realize that good testing techniques are essential to come up to these expectations. Testing tools and especially a test coverage analyser will help in this regard.

Implementing a test coverage analyser isn't a difficult task. It is important that instrumentation shouldn't affect the overall performance of the software under test and the behaviour of the source code before instrumentation. Optimal instrumentation is necessary to minimize the compiled code size after instrumentation.

Appendix A

TCA User's Manual

The test coverage analyser works just like the compiler in *Oberon*.

Open the module that must be instrumented in a viewer. If only a portion of code must be instrumented, select the portion in the text. Mark the viewer and execute the command TCA.Instrument [\s | \b] $*$.

The \s option enforces only instrumentation for *s*tatement coverage. The \b option enforces instrumentation for *b*ranch coverage and statement coverage. No option let the TCA instrument for multiple condition coverage.

The TCA will also send the instrumented file to the compiler to compile it with the \s option to generate a new symbol file. Modules that import the instrumented module should be recompiled.

When a coverage report is needed, mark the appropriate viewer with the module under test and execute the command *InstrMod*.ReportTest where *InstrMod* is the instrumented module name. The source code will be coloured according to the colour scheme in section 2.4 and a summary will be given in the System Log.

If there was a problem with the compilation of the instrumented source code, it will be logged in the System Log. This can only happen when there were errors in the original source code, or when there are multiple declarations, or if the compiled code is too long.

Bibliography

[1] Alfred V. Aho, Ravi Sethi & Feffrey D. Ullman. Compilers, Principles, Techniques, and Tools. (1986) Addison-Wesley, Reading, Massachusetts.

[2] Régis Crelier. OP2: A Portable Oberon-2 Compiler. *In Proceedings of the Second International Modula-2 Conference.* (1991) Loughborough, UK.

[3] André Fischer & Hannes Marais. The Oberon Companion. (1998) vdf Hochshulverlag AG an der ETH Zürich.

[4] JavaScope(TM) — Java Test Coverage Analyser. http://www.codework.com/suntest/jscope.htm

[5] Paul C. Jorgensen. Software Testing: a craftsman's approach. (1995) CRC Press, Boca Raton, Florida.

[6] Steve Meloan. Testing and Java(TM) Technology. http://developer.java.sun.com/ developer/technicalArticles/InnerWorkings/testing/

[7] Robert L. Probert. Optimal Insertion of Software Probes in Well-Delimited Programs. *IEEE Transactions on Software Engineering* (1982) Vol 8(1), p 34–42.

[8] Nicolas Roussopoulos & Raymond T. Yeh. SEES - A Software Testing Environment Support System. *IEEE Transactions on Software Engineering* (1985) Vol 11(4), p 355–366.

[9] Kuo-Chung Tai. Theory of Fault-Based Predicate Testing for Computer Programs. *IEEE Transactions on Software Engineering* (1996) Vol 22(8), p 552–562.

[10] Reinhard Wilhelm & Dieter Maurer. Compiler Design. (1995) Addison-Wesley, Harlow.

[11] Niklaus Wirth & Martin Reiser. Programming in Oberon: Steps beyond Pascal and Modula. (1992) Addison-Wesley, Wokingham, England.

[12] Niklaus Wirth & Jürg Gutknecht. Project Oberon: The Design of an Operating System and Compiler. (1992) Addison-Wesley, Wokingham, England.