State-Based Testing - A New Method for the Testing of Object-Oriented Programs

Christopher David Turner

Submitted as a Requirement for the Degree of Doctor of Philosophy in Computer Science

Computer Science Department,
University of Durham

12 October, 1994

Christopher David Turner

State-Based Testing - A New Method for Testing Object-Oriented Programs

Abstract

State-based testing is a new method for testing object-oriented programs. The information stored in the state of an object is of two kinds: control-information and data-storage. The control-information transitions are modelled as a finite state automaton. Every operation of the class under test is considered as a mapping from starting states to a finishing states dependent upon the parameters passed. The possible parameter values are analysed for significant values which combined with the invocation of an operation can be used to represent stimuli applied to an object under test.

State-based testing validates the expected transformations that can occur within a class. Classes are modelled using physical values assigned to the attributes of the class. The range of physical values is reduced by the use of a technique based on equivalence partitioning. This approach has a number of advantages over the conceptual modelling of a class, in particular the ease of manipulation of physical values and the independence of each operation from the other operations provided by an object. The technique when used in conjunction with other techniques provides an adequate level of validation for object-oriented programs.

A suite of prototype tools that automate the generation of state-based test cases are outlined. These tools are used in four case studies that are presented as an evaluation of the technique. The code coverage achieved with each case study is analysed for the factors that affect the effectiveness of the state-based test suite. Additionally, errors have been seeded into 2 of the classes to determine the effectiveness of the technique for detecting errors on paths that are executed by the test suite. 92.5% of the errors seeded were detected by the state-based test-suite.

Acknowledgements

There are many people to whom I am indebted, I will try and thank them all.

Firstly thanks to Dave Robson, for being my supervisor and providing me with the opportunity to study this topic in depth. Also, I would like to thank Mike Smith for his initial guidance in the topic, and for our many discussions about related issues; Ishbel Duncan and Jean Hartmann for the generous loan of numerous papers, and for numerous helpful discussions concerning various testing issues; and Ray Lewis and the VV+T group at BTL for support and funding. Also, my thanks also go to Roberto Garigliano for his suggestions concerning this thesis.

I would like to offer thanks to my parents and family for their support during my university career, and especially to my father for reviewing numerous drafts of this thesis; it is to my parents that I dedicate this thesis - no more c.t.h. !

Other people I would like to thank are my friends around the world who have kept me sane with numerous pints of beer, curries ..etc. This group includes Robert, Bod, Nick, Casey, Paul, Nigel, Jon, Annika, Nigel, Ian, and Michael.

I would also like to thank the Matthew Tebbs, for the error seeding used in the evaluation chapter, and Ian Ellison-Taylor for reviewing chapters. Finally, thanks to John Marvin who kindly lent me a machine during my writeup.

Financial support for this research has been provided by a Science and Engineering Research Council (SERC) Co-operative Award in Science and Engineering (CASE) in conjunction with BT Laboratories, Martlesham Heath, Ipswich.

This thesis has been produced using Word For Windows™ version 2.0c and 6.0c, and CorelDRAW!™ version 3.0.

™ Word for Windows is a trademark of the Microsoft Corp.

™ CorelDraw! is a trademark of the Corel Corp.

Copyright Declaration

Degree Declaration

The work contained in this thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science, and represents an independent contribution by the author. None of the work has been previously submitted by the author for a degree at this or any other University.

Contents

List of Illustrations

Figure No. **Page**

List Of Tables

Table No. **Page**

Notes on Reading This Thesis

This thesis assumes that the reader understands degree level computer science terminology. Any terms that are specific either to testing and/or object-oriented programming are defined in their respective chapters. All new terms introduced, or those with ambiguous meanings are present in the glossary (see chapter 12.) towards the rear of this thesis.

Chapter 1.

Introduction

1.1. Background

Software engineering has evolved over the past 30 years. One of its principle aims is that of quality. As projects increase in size and complexity, so does the problem of maintaining quality in the final product. The life-cycle of a product no longer consists of one stage - the implementation.

The waterfall model for the software life-cycle (see figure 1) shows the complete process of software development, starting with the requirements capture, through analysis, design, and implementation, onto testing and finally to maintenance. The software develops and grows as information is passed from each phase onto the next. The iterative nature of software production means that information is also fed back into the previous stage therefore refining the final product.

At each phase of the project, the current stage must be validated against both the original requirements and the results of the previous phase. This involves both quality assurance and Verification, Validation and Testing (VV & T). Quality assurance involves many aspects including the application of reviews to all phases of the life-cycle. Boehm [18], defines verification in terms of the question: "are we building the product right ?" In addition, he defines validation as asking: "are we building the right product ?" Many of the tasks that are performed during verification and validation are considered to be part of quality assurance.

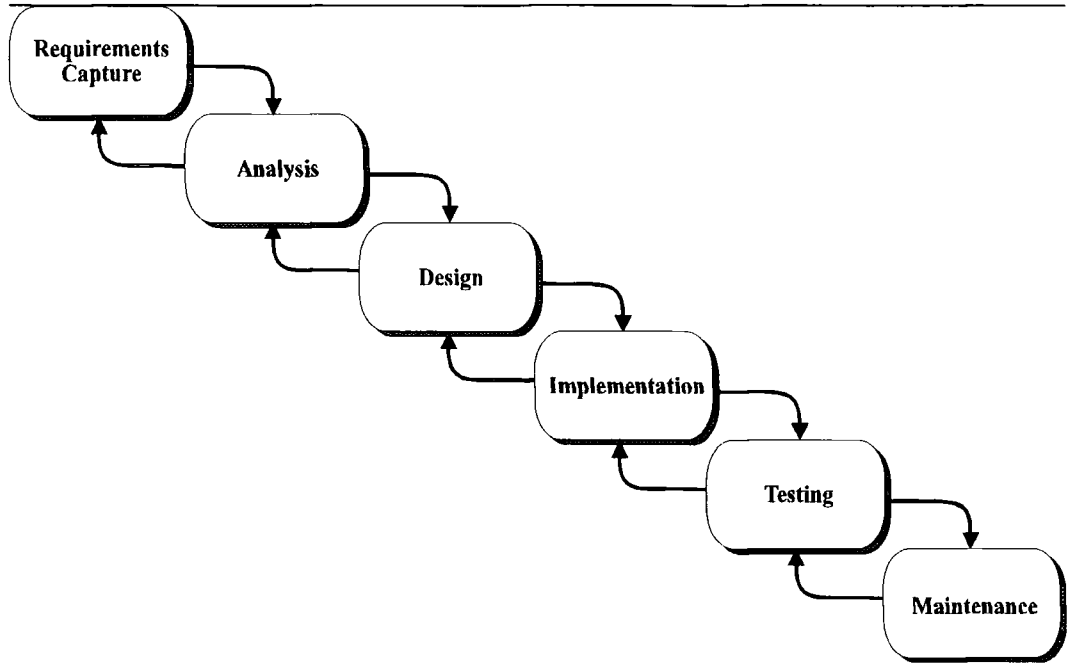

Figure 1. The waterfall model of software engineering

Testing, as the third part of VV & T is one of the later phases of the life-cycle. It involves ensuring the implementation matches both the design and the requirements of the project. There are a number of different testing techniques for traditional (procedural) programming languages. Some of these are described in chapter 3.

1.1.1. Object-Oriented Programming

Software-engineering as a discipline consists of many techniques for the production of software. Object-orientation (or object-oriented programming) is one such technique that has currently found favour amongst the software producing community. It has been used as an implementation method by small groups of people since the late 1960's. However it has recently become considerably more popular with the software industry.

Traditionally software construction was centred around the operations that were to be performed on data, whereas now more emphasis is placed on the types of data that will be manipulated. Some proponents feel that this has advantages over more traditional approaches to software production [114, 25, 100]. Within object-oriented programming, the units used for constructing software are known as objects. They embody both state and behaviour; the functionality of the object is provided by the interaction of these two.

Object-orientation is a software engineering discipline for the analysis, design and implementation of software. It is conceptually different from more traditional techniques because of it data-centred approach.

> "Object-Orientation is change in perspective to problem abstraction and software system design and development that models more closely the reality of problem domains." [25]

1.1.2. Modelling Real World Entities

The use of both state and behaviour as a single unit allows object-orientation to accurately model real world concepts from the problem domain [20, 121]. Abstract concepts are also catered for.

Encapsulation, **information hiding** and **data abstraction** are used in the construction of objects producing programs which are highly modular in nature with well defined interfaces through which objects interact with each other [29]. Encapsulation encloses the state and behaviour of an object into a single unit, forming a conceptual link between related entities. Data abstraction and information hiding are used to allow design and implementation decisions to be hidden. A change to the implementation of an object should not affect the minimum number of objects that interact with it, so long as the functionality of its interface is maintained. The use of these techniques can improve the quality of the software and hence increase the reliability of the program [43, 111].

1.1.3. Testing of Object Oriented Programs

Although a great wealth of techniques exist for the specification (for example [6, 90, 129]), analysis (for example [118, 121, 19, 20]), design (for example [20, 151, 100, 146, 57]) and implementation (for example [99, 100, 101, 133, 38, 150, 93, 94]) of object-oriented programs, little research has considered the VV&T phase of the software lifecycle for object-oriented programs. Current research in this area falls into the following categories:

- theoretical aspects of testing of object-oriented programs
- the use of formal methods to prove an object's conformance with its specification
- the derivation of test cases from a formal specification
- the adaptation of traditional testing techniques to object-orientation
- testing of classes defined using inheritance

Formal methods however, are difficult to use and require an experienced mathematician. At present there is a lack of theorem provers and tools for aiding the mathematician during the development and validation of formal specifications and the objects derived from them. Without the use of automated tools there are opportunities for human errors to occur [45]. To avoid these, extensive checking of the reasoning used in the derivation of proofs must be performed [7]. Formal specifications also have difficulty in capturing non-functional requirements such as performance and visual aspects. It is for these reasons that software testing is still a viable and effective alternative to the formal verification of software components.

The adaptation of traditional approaches (see chapter 6.) provides a wealth of techniques which are potentially applicable to object-oriented programs. However, as they were originally intended for procedural programs, they are unlikely to validate those aspects of programs that are peculiar and unique to object-oriented programming. It is for this reason that a new technique is required to validate object-oriented programs. One aspect not directly addressed is the interaction of an object's behaviour with its state. In this thesis a new technique known as state-based testing will be described. State-based testing complements other techniques available by validating the interaction between the behaviour of an object and its state.

State-based testing models objects as **finite state automata** (see chapter 5.). Finite state automata are mathematical entities based upon a current state and a list of possible transitions. An automaton's next state is only dependent upon its current state and the sequence of the applied inputs. By modelling classes as finite state automata, predictions can be made about the behaviour of the classes under specific circumstances. Classes can then be tested by predicting the state transitions based upon those that take place in the model. Detecting whether an object is in the correct state, or if it is in a valid state is then a relatively simple process.

1.1.4. The Testing Process

Testing is applied at different stages during a project's lifecycle. Two of these, **unit testing**, and **integration testing** are directly affected by the technique used for the program's implementation. Unit testing is the validation of individual construction units of a program (usually functions or modules) in isolation. Integration testing then combines these units and validated that they cooperate correctly to achieve the required functionality. This thesis is concerned primarily with unit testing, although for completeness, state-based testing is also discussed with respect to integration testing (see chapter **6.**).

At the unit phase, the focus is on the state transitions that take place within the class under test, whereas, the integration testing phase must focus on the external state transitions caused by the class under test. State-changes must be detected in all external objects, including: parameter objects, all global objects, and any other objects that the class under test has access to.

State-based testing has been designed to validate a particular aspect of object-oriented programs. Hence it must be applied in conjunction with other techniques to provide complete validation of object-oriented programs. However, an extensive discussion of complementary testing techniques is beyond the scope of this thesis.

To extend the application of state-based testing to include classes that have been defined using inheritance, the technique has been integrated into the incremental testing algorithm of Harrold et al. [55]. The algorithm uses the dependencies between the operations and the attributes of each class to determine which inherited operations must be retested in the new context of the child class. This analysis is enhanced and clarified by the analysis of the states involved. This is discussed further in section **6.4.**

1.2. Outline of Thesis

All terms that are required for the description of state-based testing are introduced in the chapters following this introduction. Chapter 2 briefly introduces object-oriented programming and the terms associated with it. Chapter 3 outlines some of the concepts associated with software testing. For consistency this thesis will adopt one set of terminology that will be used to refer to particular concepts of both object-oriented programming and software testing. Although some readers may be familiar with these topics, these chapters clarify the terms that are used throughout the remainder of this thesis and hence should be read even if only superficially.

Where appropriate, the text has been interspersed with examples to clarify various aspects of the technique being described. Source code examples are written in the syntax of the C++ language, even though they are applicable to many other object-oriented languages.

The remaining chapters of this thesis are as follows:

[Chapter 2] "Object-Oriented Programming" introduces and discusses the major concepts and issues of object-oriented programming.

[Chapter 3] "Software Testing" introduces and describes different aspects of the testing of traditional programs. The vast majority of these concepts can also be applied to the testing of object-oriented programs (this is discussed later).

[Chapter 4] "Testing Object-Oriented Programs" reviews the currently available literature covering the testing of object-oriented testing. The chapter is organised by concept, rather than by chronological order. The previous 2 chapters are summarised in addition to chapter 4. This information is then used to provide a set of requirements for a new technique. State-based testing is assessed against these requirements in chapter 10 (Conclusions) to demonstrate the techniques suitability for testing object-oriented programs.

[Chapter 5] "A New Technique - State-Based Testing" describes the theoretical background and practical application of the new technique. The technique models objects as finite state automata which are then used for the generation of test cases. The basic technique is extended with the additional concepts of scenarios and these are integrated into the testing process. Any exceptions to the technique are noted.

[Chapter 6] "The Testing Process" first discusses the adaptation of traditional testing techniques to the testing of object-oriented programs. This is then integrated into the notion of grey-box testing whereby a combination of techniques are used to provide a greater level of validation for software. State-based testing is integrated into the traditional notions of unit and integration testing and the benefits of this integration are discussed. Finally, the potential use of state-based testing in conjunction with other techniques such as the incremental testing techniques of Harrold et al. is also discussed.

[Chapter 7] "A Suite of Prototype Tools" describes the tool suite developed for the evaluation of the state-based testing technique. Three tools are outlined, the first MKTC uses tester supplied information to generate the suite of test cases. The second, MKMFTC aids in the automatic compilation and linking of the test cases, and the third, TESTRUN automates both the execution of test cases and the gathering of results.

[Chapter 8] "Case Studies" details four classes that have been chosen for the evaluation of the technique. For each class, a description of its background and a summary of its

design is provided. The code coverage of each class executed by its test suite is also provided in addition to an analysis of the remaining unexecuted portion of the class.

[Chapter 9] "Evaluation of State-Based Testing" uses the results from the four case studies in the previous chapter to discern some of the factors that affected the level of each class executed. Two of the case studies are then seeded with errors to evaluate the effectiveness of the technique for discovering errors in portions of a class that has been executed by the test suite.

[Chapter 10] "Conclusions" summarises the information presented in this thesis. A critical analysis of the results is provided in addition to some of potential directions for future research into the state-based testing technique. The requirements for a new technique that are outlined at the end of chapter 4 are used to assess the technique.

[Chapter 11] "References" contains an alphabetical (and numerical) list of the references cited within this thesis.

[Chapter 12] "Glossary" contains words that are defined in the thesis. It also contains definitions for words that are ambiguous, or rarely used.

[Appendix 1-5] "Appendices" contains additional information that some readers might find helpful in understanding and following the information outlined in this thesis, including the source code for the 4 case studies.

Chapter 2.

Object-Oriented Programming

2.1. Introduction

This chapter will introduce the basic concepts and terms associated with object-oriented programming and different object-oriented programming languages. Different facilities provided by each object-oriented programming language have different terms associated with them even though they often refer to the same concept. For the purpose of this thesis one set of terms will be chosen, highlighted and used throughout the remainder to reduce any ambiguity.

Some of the early history of object-oriented programming is introduced first. This is followed by a definition of an object and all that constitutes it. A discussion is presented which outlines the definition of an object-oriented programming language. This is supplemented by a discussion of each type of language and the facilities that are indicative of it. Those concepts that are not within the scope of state-based testing are noted as such.

2.2. History

Object-Oriented programming is the construction of programs as a collection of interacting **objects**. Simula-I [33] (which later became Simula-67 [15]) was the first language to support this new

philosophy. Simula introduced the concept of a **class** which was a template encompassing both a **data-structure** and a set of routines to manipulate the data-structure. Objects are created by **instantiating** the class. Simula also provided **prefixing**, a method for defining new classes from old ones by inheriting the data-structure and the routines. This technique was later to be known as **inheritance** (see section 2.4.4.2.).

The concepts introduced by Simula were later incorporated into the various versions of Smalltalk [80, 117]. In Simula all objects have a type (effectively the class from which it was created), however, Smalltalk objects do not. Typed languages, use the type of an object to restrict the operations that can be performed on it. Both Smalltalk and Simula belong to the category known as class-based Object-Oriented Programming Languages (class-based OOPLs) which are discussed in section 2.4.3.1.

During the 1970's and early 1980's, a number of other languages were developed which incorporated some if not all of the concepts of Simula, Modula-2 [152] and Ada [144]. Many of these only provide constructs for modularisation and data abstract and hence these are generally not considered to be Object Oriented Programming Languages (OOPLs). This is discussed further in section 2.4.

2.3. What is an Object ?

An object is defined by Cato and Warren as an entity which has:

> "characteristic attributes that describe the way it is AND characteristic behaviour that describes what it does." [25]

This notion is extended by Booch:

> "[an object has] state, behaviour and identity; the structure and behaviour of similar objects are defined in their common class; the terms instance and object are interchangeable" [20].

Each of these concepts will be described in the sections below.

Objects are the central concept for the construction of object-oriented programs. They are **run-time** entities incorporating the notions of information hiding, encapsulation and data-abstraction in their construction. The term 'information-hiding' was first coined by Parnas [110] in the early 1970's. He used it to describe that access to a module must take place through a well defined interface. All other detail should be hidden from view and any attempt to access it should be disallowed at compile time.

Encapsulation is a method by which information hiding can be realised [79]. Whereas, data-abstraction is the modelling of a concept as a black-box with a well defined interface through which

interaction takes place. Any implementation detail is hidden from view, allowing design decisions to be changed without affecting users of the abstraction [124]. The abstraction provides the facilities for interaction with the object's state in a controlled manner [125]. Through the facilities provided, the abstraction also controls how other objects in the system visualise the data that it represents. All these concepts contribute to a tightening of the **coupling**[1] between the state and the associated operations [84].

Object-oriented programs are generally highly modular by nature with low coupling between objects, but high coupling within objects. An object (A) should only access another object (B) via the interface provided by B and should do so with no regard for the implementation of B [39, 125]. The independence of objects from each other's implementation eases both the development and maintenance of the software [114]. Additionally, the construction of a program in this manner more closely models the entities in the application's problem domain (the program's requirements).

2.3.1. Identity

The **identity** of an object:

> ".. is that property of an object which distinguishes it from all other objects." [81]

it has two main aspects, durability and distinguishability. durability means that no matter what values change within an object, it is still the same object we are dealing with. Distinguishability allows the detection of one object from another [81]. Identity in OOPLs is achieved in part by the use of unique variable-names associated with individual objects, in addition, each object has its own unique address in the application's address space. Identity is an aspect more usually associated with objects within object-oriented databases [82], although it is directly associated with objects in Emerald [16].

2.3.2. State

The Oxford English dictionary defines **state** as an:

> "existing condition or position of thing or person" [107]

---

1 Coupling is loosely defined by the location and number of variables that are used within a class which are not directly passed as parameters.

Hence, the state is the 'memory' that persists between the invocation of the various **operations** provided by the object [146]. It is used for both storing values and for communicating events and situations from one operation to the next. The portion of the state used for communicating events and situations will be referred to as the **control-information** portion, and the remainder of the state will be referred to as **data-storage**. Both of these terms will be used frequently during the remainder of this thesis.

In programming terms, the state of an object is the combination of the value of all the **attributes** that form part of the object [118, 96]. An attribute is an individual variable that forms part of an object's data structure. It is the abstraction of a characteristic or property of the object [121, 95]. Attributes are also known as **data-members** in C++, and **instance variables** in Smalltalk. These terms are interchangeable, however, for consistency only attribute will be used within this document.

There are three main types of attributes provided by OOPLs:

- instance attributes, these are local to objects. Every object has its own copy of these.
- Class attributes, these are local to classes. Classes define a template from which objects are created (class-based OOPLs are discussed in section 2.4.3.1.). Only one copy of class attributes exist per version of the class in the system. All objects created from the class have access to them.
- Global attributes, only one copy of these exist in the whole system. All objects have access to these irrelevant of which class the object was created from.

2.3.3. Behaviour

The behaviour defines how the object will function. An object provides a set of interface routines through which all interaction must take place. These routines manipulate the state, and collectively they define the object's behaviour. The routines associated with an object are known by many different names for different OOPLs. In C++ they are known as **member-functions**, in Eiffel they are known as **features**[2], in Smalltalk they are known as **methods**, and in Trellis [83] they are known as **operations**. These terms are interchangeable, but for consistency the term *operation* will be used to describe a routine associated with an object.

---

2 The term 'feature' is actually the term for anything exported as part of a class interface, in Eiffel. Therefore, it includes any attributes that are exported by the interface.

Each operation is invoked by sending a 'message' to an object requesting it to perform a service, the object then invokes the appropriate operation for the message received. This is conceptually different from the invocation of a procedure or function which normally infers an action is performed instead of a request for a service [117]. The calling object does not care about how the receiver is to satisfy the request, only that it is satisfied.

All messages have two associated objects, the sender and the receiver [146]. Messages may pass objects as parameters, and they may return objects as results.

All object provide at least two operations, the **constructor** (or create operation) and the **destructor** (or destroy operation). The constructor is the operation that is used to create an object and initialise it to a known state. By convention, an object may provide many different constructors, although it usually only provides one destructor. The destructor is the operation called to destroy an object, it performs any tidying up required before returning allocated memory back to the system.

2.3.4. Message Passing

Parallel programming languages also use a message passing technique for communication between processes. Hence a comparison can be drawn between the process and object-oriented paradigms [130].

Both processes and objects respond to messages, however the vast majority of OOPLs are sequential not parallel, hence objects cannot initiate communication asynchronously as the thread of execution follows the messages being passed. Therefore there is no likelihood of deadlock or other conditions that blight parallel processes.

These issues are not addressed by state-based testing which is design to validate the state-transformations that occur within objects, currently the technique only addresses sequential OOPLs, that is, objects that can only execute one operation at once.

2.4. Object-Oriented Programming Languages

The precise definition of what constitutes an OOPL is not clear. Supporters of each language purporting to be 'object-oriented' provide a slightly different definition. However, all parties agree that

the language must provide a mechanism for data abstraction [30, 134, 111]. This debate is clarified by Stroustrup who defines an OOPL as

> "... a language that has mechanisms that support the object-oriented style of programming well." [134]

This infers that although programs can be written in an 'object-oriented' style in the vast majority of languages, an OOPL is a language that provides support for the object-oriented style of programming. A basic model for OOPLs is provided by Nguyen and Hailpern. In it, objects communicate by sending messages to each other. If an operation is inherited from another object, then upon receipt of a message for it, the message is immediately re-despatched to the parent that actually defines the routine. This model does not define any notion of object creation therefore avoiding the class/prototype discussion (see section 2.4.3.).

There is no definitive answer to which languages are OOPLs; languages which conform to the class-based definition will not conform to the same principles as a prototype-based OOPL. Hence, a separate definition will be provided for each type of language. Any language that does not fit any of the categories described later will (at present) not be considered an OOPL.

All OOPLs fall into one of two main categories, **pure OOPLs** and **hybrid OOPLs**. Pure OOPLs are languages which have been developed from first principles to embody the object-oriented principles; hence they do not contain constructs for any other development method. In contrast, hybrid OOPLs are languages which have been developed by extending a previously existing non-object-oriented programming language.

2.4.1. Pure OOPLs

Eiffel [100, 102], Smalltalk [46, 72] and Self [143, 26] are examples of pure OOPLs. Eiffel was developed with the requirement for a language which supports sound software engineering principles, hence it incorporates strong typing, and assertions. Strong typing, is the checking by the compiler and the executing program that the types of objects passed as parameters to operations are of the correct type. A type specifies the operations that an object will possess and that can be called either from outside or inside the object. Type checking performed on parameters ensures that all the objects passed provide the necessary operations that will be required. More errors are caught at compile-time with strong typing [43]. The compiler is then under an obligation to generate code that will not fail because of mismatching types when executed [49], although it can still fail due to other errors.

Assertions are statements about the values of data within the program. They are used to check the validity of parameters being passed to routines (**preconditions**) and any return values (**postconditions**). Also assertions may be made about the validity of an object's state before and after a call (an **invariant**). These facilities form part of the conceptual 'programming by contracts' that is an integral part of Eiffel [100, 101] ([91] describes assertions for other languages). A contract declares that if a routine is passed parameters that satisfy the precondition, then the routine will in turn return values that satisfy the postcondition.

2.4.2. Hybrid OOPLs

In contrast to the strongly typed class-based OOPL Eiffel, Self provides the ability to declare prototype objects. These are discussed further in section 2.4.3.2.

Examples of Hybrid OOPLs are CommonObjects [126], CommonLoops [17], Oaklisp [86], Objective-C [32] and C++ [133, 38]. CommonObjects, CommonLoops and OakLisp are all derived from the language Lisp, although each provides different facilities and constructs. CommonObjects provides pseudo-variables which allow operations to look and act as if they are actually variables, whereas CommonLoops places great emphasis on meta-objects, that is, objects that define objects.

In contrast, both Objective-C and C++ are based upon the language C. Objective-C is a combination of two languages, C and Smalltalk. C and Smalltalk have quite different philosophies, C was developed for speed and efficiency, whereas Smalltalk was developed for its expressive power. C++ on the other hand, was developed with a similar philosophy to C, that of efficiency with the added benefits of strong typing and the use of the object-oriented technique for software construction. C++ originated from early work by Stroustrup on the addition of a class construct to the language C [132]. C++ has gained a large following in the software construction community and compilers for it are widely available. It will therefore be used for the examples throughout the remainder of this document.

2.4.3. Language Types

Using another categorisation, there are three major types of OOPLs: class-based languages, prototype languages and Actor languages. These reflect how objects are modelled and created in the language. Each type will be discussed below.

2.4.3.1. Classes

Class-based languages form the most popular single group of OOPLs available. A class defines a template from which objects are created, describing the attributes and the behaviour of all instances of the class. All objects of a given class are identical in the attributes and behaviour they possess, although the attribute may contain different values; all objects must be associated with a class.

A class is similar to the package concept in Ada, and the

The vast majority of languages provide two sections to the class declaration: the interface, and the main body of the class. The interface defines the operations that are provided by any objects created from the class, and any messages they can respond to. All objects created (or **instantiated**[3]) from a particular class will respond to the same set of messages. The body is used for the declaration of the code for the operations.

When declaring a class, there are three areas in which to put the declarations of operations and data. These will be known as the **private**, **protected** and **public** areas[4]. All class-based OOPLs provide at least one of these.

Any operations or data declared in the public area are accessible to the operations of any other objects, hence it is not advisable for data to be declared in this area as its integrity cannot be assured between calls of operations [141].

Declarations in the protected area are only accessible to operations of the same class and any class derived from it (**derived classes** and **inheritance** are discussed in section 2.4.4.2.). As derived classes are extensions or adaptation to a parent class, any access to data in the protected area is more likely to be justified. Hence data integrity is potentially greater than with the public area.

The private area provides maximum data integrity, with access only being granted to operations of the same class. However, this degree of restriction of access to the data of the class reduces its extendibility and its adaptability.

---

3 to *instantiate* means "to create an insteance of."

4 These are the same names as used by the C++ language, although the concepts appear in many object-oriented languages.

The popular definition of the facilities that must be provided by a class-based language for it to be categorised as an OOPL are [58, 111, 25, 84]:

1) object classification including specialisation and adaptation
2) polymorphism (see section 2.4.5.)
3) dynamic binding. (see section 2.4.6.)

Object classification is achieved by all objects belonging to the classes from which they were created. This also implies that the language actively provides a construct for the declaration of classes. A language supports specialisation by providing a construct such as **inheritance** or **delegation**. These will be discussed in sections 2.4.4.2. and 2.4.4.3. respectively.

2.4.3.2. Prototypes

An alternative method for creating objects is by **cloning**. This is the basis of prototype-based languages. Instead of defining a class from which sets of similar objects are created, a prototype-object would be created. A prototype is a standard example instance of an object [21].

New objects are cloned from the prototypes and then adapted. The adaptations can include the addition/removal of attributes and/or the addition/removal of operations. A system built using prototypes is more likely to consist of a large collection of unique objects, whereas a system built using classes would consist of collections of identical objects. Programming with prototypes is sometimes referred to as differential programming.

Some proponents believe that the use of prototype-based languages allows a design to be expressed in code more easily especially for defining windowing systems[5], however, because of the lack of a classification mechanism (classes) it is more difficult to handle abstract concepts such as integers [21, 143].

---

5 Windowing systems are also known as Graphical User Interfaces (GUI).

2.4.3.3. Actors

The Actors group of languages are a special type of prototype-based OOPLs. Actors use fine-grain parallelism[6]. Whereas the majority of class and prototype-based OOPLs are either executed serially[7], or use the object or thread as the lowest level of parallelism. Actors are

> "computational agents that carry out their actions in response to incoming communications" [1]

A program is a community of communicating actors, some of which may communicate with the outside world. An actor is defined by a mail address (to which messages are sent), and a behaviour. An actor has the following kinds of actions:

- send communications to other actors including itself,
- create new actors,
- specify a replacement to answer the next message

Actors, however do not use the assignment operator '='. Instead they use a technique known as replacement. This it is claimed, avoids the so called Von-Neumann bottleneck (see [5]).

Because of the fine-granularity of the parallelism inherent in the Actors model of programming, the technique to be described in this document is not applicable. State-based testing is dependent upon the serial nature with which most object-oriented programs are executed.

2.4.4. Reuse Mechanisms

Object-oriented programming provides facilities to enable an increase in the reuse of software components. Four mechanisms are available for enabling reuse:

(1) instantiation or aggregation

(2) generic classes

(3) inheritance

(4) delegation

(1) is provided by all OOPLs. Aggregation is simply the combining of objects to form new objects. (2) - (4) are discussed below.

---

6 Fine grain parallelism is parallelism at the statement and substatement level.

7 Although parallel extensions to a number of OOPLs have been produced.

2.4.4.1. Generic Classes

A **generic class** is a class from which other classes are created. One or more of the objects required have unspecified types which can be supplied as parameters when a class is to be created. For example, a generic list class could be supplied with a type allowing the creation of 'a list of' integers, or 'a list of' filenames'.

The term generic classes is used by the Eiffel community [99, 100, 3], whereas they are known as templates in C++ [38, 3], and meta-classes in Smalltalk [46].

2.4.4.2. Inheritance

Inheritance is a technique for adapting, extending and specialising classes for a new purpose. It can be used for factoring out common concepts, code and data into a class from which other classes are defined [153]. Although inheritance is usually associated with class-based OOPLs, another type of inheritance is based on the specialisation of objects (rather than classes). This is discussed both by Sciore [120], and Ungar et al. [143].

The two main uses of inheritance are:

- defining a type hierarchy (known as **specification inheritance**)
- code reuse (known as **implementation inheritance**)

The type hierarchy is used in conjunction with polymorphism (see section 2.4.5.) to define the different ways in which objects of different classes can be legally substituted for each other. The type hierarchy therefore defines the messages that objects of each class can respond to.

The class that is being extended or adapted by inheritance is known as the **parent** or **base class**. The new class that is being defined is referred to as the **derived** or **child class**. For consistency the terms *parent* and *child class* will be used throughout the remainder of this thesis.

The use of inheritance for code reuse has sparked a great deal of controversy over its uses and abuses. Inheritance should be used to define an 'is-a' relationship between the child and the parent classes [99, 51, 85].

Some OOPLs only allow child classes to be defined from one parent class, this is known as single inheritance. Inheritance from more than one parent is known as multiple inheritance. Formal semantics for multiple inheritance are defined by Cardelli in [24].

In a large number of OOPLs, the child class has access to the representation declared by the parent class. This is enabled by declaring the attributes of the parent class in the *protected* area of the class declaration. However, this bypasses the encapsulation of the class and reduced the integrity of the inherited state [104]. The integrity can be preserved by the use of pseudo-attributes which insulate any child classes from alterations to the implementation of the parent class. They also allow any interaction between the child class and the inherited part of the state to be validated by the parent class [124, 150].

In certain circumstances, a parent class can be used to define a common interface for all its child classes, without providing some or any of the implementation. Operations that are declared with no implementation are known as **deferred** or virtual operations. Objects cannot be created from a class that has at least one deferred operation; these classes are known as abstract base classes. Objects can only be created from a child class whose parent is an abstract base class, if it provides implementations for all deferred operations.

Operations that have the same name are known as overloaded operations. The compiler chooses the correct operations by the type of the parameters that must be passed.

2.4.4.3. Delegation

Delegation is a mechanism by which one object can delegate the responsibility for performing a task to another object. These tasks are performed in response to messages. When the responsibility for answering a message is delegated to another object (the delegatee), a reference to the original recipient (the delegator) of the message is also passed. The new recipient can then query the delegator if more information is required [153]

Delegation can be both a static and a dynamic facility. Delegatees can be specified in an objects interface. In addition, indices into arrays of delegatees can be used to dynamically change the new recipient of a message [2].

A draw back with delegation is the lack of classification when compared with that achievable with the use of inheritance and classes. Although the lack of a classification allows delegation 'freedom of expression', it does not provide structural guarantees for the program as an object may be able to respond to different messages at different points in its lifecycle. Hence, both techniques have their advantages. Stein [128] demonstrates that delegation can model inheritance and vice versa by

mapping prototypes into classes. Delegation can be implemented in class-based OOPLs [77], although it is generally not provided by them.

State-based testing is not currently applicable to objects defined using delegation.

2.4.5. Polymorphism

Polymorphism simply means "many forms". A polymorphic reference is one that will point to instances of different classes during the programs execution. In a large number of class-based OOPLs, this substitution is restricted by the inheritance hierarchy. A class can only be substituted for its parent classes because it responds to the same messages. In Smalltalk however, objects do not have types allowing any substitution to take place, the 'message not understood' error is provided for the case when a message is sent to an object that cannot answer it.

2.4.6. Dynamic Binding

Binding is the process of associating an identifier (the name of the operation to be called) with the implementation of the operation provided by an object (the implementation). Some languages allow some binding to be done at compile time (for example, C++ and Eiffel) and is referred to as **static binding**. Binding performed at run-time is referred to as **dynamic binding**. This facility is used in conjunction with polymorphism allowing different implementations of an operation to be called depending upon the type of the current object.

Polymorphism is not directly addressed by state-based testing.

2.5. Summary - Applicability of This Thesis

This chapter has outlined the basic concepts that are part of object-oriented programming.

The basic unit of object-oriented programming is the object. They are created, manipulated and destroyed during a programs execution. State-based testing addresses all three of these phases. It involves the careful analysis and validation of the current state that an object possesses. However, the Actor group of languages introduce the use of fine-grain parallelism. This alters the notion of

sequential transformations being performed on a persistent store[8]. State-based testing is based upon a model of the sequential transformation, and hence cannot be used with Actor languages.

State-based testing is applicable to languages that statically use inheritance to extend and adapt classes for new purposes, this is discussed in more detail in section 6.4. State-based testing is also applicable to objects defined with prototype-based OOPLs, however because of the decentralising nature of delegation, it is not applicable to the validation of objects that delegate to other objects.

Polymorphism is not directly addressed by the technique as polymorphism is concerned with the dispatching of messages to the correct implementation, whereas state-based testing is concerned with the transformations that take place within the state of an object, cause by the code of a specific operation.

---

8 That is, the 'memory' of the object.

Chapter 3.

Software Testing : An Overview

3.1. Introduction

Validation, Verification and Testing is a necessary activity that takes place throughout the software lifecycle. Testing is the phase that is applied to the implementation of the project. It can be applied at both the unit and integration phase where test stubs and drivers are used to emulate missing or unvalidated portions of the project.

Testing techniques are categorised into two main sections, black-box and white-box. Black-box techniques treat the code under test as a black-box with inputs and outputs, test cases are derived from the specification alone. In contrast, white-box techniques derive test cases from both the code and the specification.

This chapter introduces testing by first describing its place in the software lifecycle (section 3.2.). This is followed by a discussion of both unit and integration testing (sections 3.4. and 3.5. respectively), both of these sections outline many of the various techniques that can be applied at that level.

3.2. Testing in the Software Life Cycle

As mentioned in chapter 1, software projects follow a software lifecycle which describe the conceptual development of the system from its inception at the requirements phase, through to its delivery, and on to its maintenance. Activities performed early on in the software lifecycle cannot be guaranteed and hence there is a requirement for testing to validate that the software conforms to its design and specification.

The definition of testing has been adapted from the IEEE standard by Smith and Robson, they define testing to be:

> "the process of executing a program with the intent to yield measurable errors. It requires that there be an oracle to determine whether or not the program has functioned as required, with comparison of performance against a defined specification." [123]

Testing is a destructive process designed to locate errors in a program [106]. The errors are introduced because of human error on the part of the programmer, or the system specifier/designer. Errors may exist in the form of design faults leading to an incorrect system architecture, or incorrect assumptions by the program causing code not to conform to its specification. Hence there is a need for testing throughout the lifecycle, not just after the implementation phase.

The process of applying testing throughout the software lifecycle is known as Verification, Validation and Testing (VV & T). Verification ensures that the software is being built correctly by testing the products of each phase against the previous one. Validation is used to ensure that the correct product is being built by constantly checking each lifecycle phase against the original project requirements. See [145] for more details.

Testing techniques can be categorised either as **black-box** techniques, or **white-box** techniques. Black-box techniques consider the unit or system under test as an entity with a mapping from its inputs to its outputs. Only the external behaviour is considered. In contrast, white-box techniques use information obtained from the program's logic for the derivation of test cases.

Testing is applied at a number of different levels. **Unit testing** tests the individual modules of a program in isolation. In contrast, **integration testing** tests units in co-operation with each other. White and black-box techniques can be used for these phases. **System testing** exercises the whole program as a single unit. This testing phase is performed with a black-box view of the system. Its aim is to confirm that the system functions as per its requirements. The final phase is **acceptance testing**.

Acceptance testing is performed by the customer, they perform tests to ensure that the software adheres to *their* requirements.

All forms of testing require testing resources. Testing resources are measurable entities that are finite in quantity and (usually) have an associated cost. This includes the time used by testing personnel whilst testing a project, along with the machine time and disk space required to generate and execute the test cases. Testing resources must be scheduled to maximise their use. If a technique is automated, it requires fewer testing resources to accomplish the same level of testing and hence allows more testing to be performed [14].

3.3. Test Oracles

Testing is performed by creating an initial scenario, exercising the unit or program under test, and verifying the results obtained. The verification is done by using a **test oracle**. A test oracle is a person or program that is able to determine whether the output or results generated by a test case represent the expected values. Program-based test oracles can consist of many forms including Prolog programs [131], simple output file comparisons, or programs able to use formal specifications [42, 75, 4].

3.4. Unit Testing

The first form of testing performed on a program is unit testing. It involves the testing of individual units of a program in isolation. The size of unit is usually the smallest unit in the design which is usually either the function or module.

Each unit to be tested in isolation requires the use of **test stubs** and **test drivers**. Test stubs are replacements for units that are *called* by the unit under test. Their purpose is to provide a minimal simulation of the modules they replace and to allow the tester to examine the values passed as parameters. It is not always practical to provide a simulation, especially when complex data structures are involved, in which case the actual modules must be used.

Test drivers are routines written to provide the correct values as parameters to the module under test. They must create the initial scenario for the test case, execute the test case, and validate the final results. Sometimes it is possible for results to be automatically validated, however if this is not possible the test driver must refer the validation of the results to a human oracle.

There is an cost overhead involved in the development of test stubs and test drivers. This overhead is kept low by providing test stubs which are only simple simulations, or which question the tester for the required values.

The testing techniques that can be applied at the unit level fall into two basic categories, black-box techniques and white-box techniques. Some of these will be described below.

3.4.1. Black-Box Testing

Black-box testing (also known as functional, or specification based testing) is the testing of a unit against its defined specification. The unit's structure is completely ignored therefore the unit is treated as a black-box. Only the unit's specified behaviour is used for the generation of test cases.

Functional testing techniques rely heavily upon the completeness of the unit's specification. A program source code is usually readily available, however complete functional specifications are rare. Functional specifications can be written informally in natural language, or formally specified using mathematics.

Functional testing techniques are described below (see [70, 106, 9, 31] for more details).

3.4.1.1. Random Testing

Random testing involve the random generation of values to be used as inputs for the unit under test. Using probability, this technique has the least chance of discovering errors in a program [106]. However some proponents still believe it is still a viable testing technique [37].

3.4.1.2. Equivalence Partitioning

Equivalence partitioning is one of the simplest black-box techniques. The unit's specification is analysed for parameter values and ranges of parameter values that are expected to be treated similarly. These values are group in equivalence partitions. All values in a partition are expected to be treated similarly and so are equivalent to each other for the purpose of testing. A single value is chosen to represent each partition [106]. Partitions should be included for erroneous values as well as valid values. The technique reduces the potentially infinite range of values down to a manageable set of partitions.

The technique is only as good as the information that is used to define the partitions. Hence, it is heavily dependent upon the completeness of the unit's specification. No design or algorithmic information is used to determine the partitions and so the technique is unlikely to cause the execution of all the paths in the program.

This basic black-box technique can be extended to include partition information from the design and the source code. This enhancement is described along with other white-box techniques in section 3.4.2.

3.4.1.3. Boundary Analysis

Equivalence partitioning is used to determine equivalent partitions of input parameter values. Boundary analysis extends equivalence partitioning by selecting more values to represent each partition. Instead of choosing a single value to represent the partition, values are chosen which are near to the boundaries of the partition along with a value in the centre of the partition. This is performed for partitions which represent erroneous values as well as valid values.

Boundary analysis is also applied to the values expected as output from a unit. Test cases are generated so that at least one test case generates boundary values in each output partition.

3.4.1.4. State-Transition Testing

The software component under test is analysed for conceptual states, for example, a button can be up or down. The processes that are required to change the component from one state to another is analysed and this information is used to model the component as a finite state machine. Often the specification of a component contains the required information for this modelling to take place.

Test cases are then taken from the model, such that the component is exercised through all its possible transitions to ensure that it performs correctly. The main problem with the technique is the analysis of the conceptual states. This is sometimes a very subjective decision and hence not very rigorous.

This technique is reviewed in more depth in the next chapter.

3.4.1.5. Cause-Effect Graphing

Combinations of parameter values can be explored using a technique known as cause-effect graphing. The specification is broken down into pieces of a manageable size [106]. The pieces are analysed for **causes** and **effects**. Causes are significant input conditions or input partitions. Effects are significant output values or partitions. The causes required to generate each effect are then mapped on a graph and a table, and used to generate test cases.

3.4.1.6. Design-Based Testing

A program is broken down in to a number of distinct functions within the design. Some of these functions will be reflected directly in the code, whereas others will not. The functions that are reflected directly are known as computational functions because of the type of task they perform. Functions that are not directly reflected are usually control-functions.

Traditional functional testing is applied at the system level to the program as a whole. Test cases are derived from the application of various heuristics to the input domain of the program. The program's function is validated by comparing the values returned with those expected.

The program is viewed as a collection of functions. Design-based testing involves the careful choice of test data so that each and every design-function is exercised. This is performed by applying the traditional functional techniques such as equivalence partitioning and boundary analysis to the input domain of each design-function. This technique is highly applicable to numerical and scientific programs which deal with complex calculations [68].

3.4.1.7. Error Guessing

A combination of experience, intuition, and information about the type of unit being tested can be used to generate test cases designed to detect particular types of errors. This technique involve the identification of possible assumptions that might have been made by the unit's author.

3.4.1.8. Limitations of Black-Box Testing

Although units are ultimately written to meet their requirements, testing a unit against its specification alone is not sufficient enough to guarantee that all errors present in the code have been located [106, 154, 60]. Black-box testing cannot even guarantee the execution of all statements in the

unit. Additionally, as test cases are derived from the specification alone (in the majority of cases), this does not allow for the inclusion of test cases which test the algorithmic nature of the unit. These points are addressed by white-box testing techniques described below.

3.4.2. White-Box Testing

White-box testing (also known as structural, or program-based testing) is the testing of software using information about a program's structure. Test cases are derived by careful analysis of the program's executable paths and logic. The techniques can be categorised into two groups, static techniques and dynamic techniques. The majority of dynamic techniques measure the coverage of a program (the amount of the program exercised) and determine which new test cases are required to executed the unexecuted portion. Static techniques (except symbolic evaluation) involve the tester visually checking the program.

3.4.2.1. Static Techniques

Static techniques do not involve the execution of the program. Hence, no 'test cases' are actually developed although the program may be checked against various criteria either manually or automatically.

3.4.2.1.1. Code Inspection and Walkthroughs

This is also known as 'human testing'. It involves the tester using various criteria to work through the program. The checking is normally performed by small groups of people with the aid of the program's author. The testers ask the author questions if the intent of certain parts of the code is not self-explanatory. Code inspection by the author alone is a far inferior technique because of the author's implicit assumption that the code works. Myers notes that as testing is a destructive process, an author is less likely to destroy their own work than other people are [106].

Walkthroughs involve a group of people manually tracing the execution of the program. Both walkthroughs and code inspection tend to detect the actual error in the program whereas dynamic testing only reveals the symptoms of errors.

Experience has shown that manual checking of programs is a very effective testing technique. They can discover between 30% and 80% of all errors that will ever be detected in the program [106].

These techniques are discussed in more detail in [106, 40].

3.4.2.1.2. Symbolic Evaluation

Symbolic evaluation is the execution of a program by evaluating the symbols and expressions used. Variables can be used to store either values or symbols. If symbols are used, the evaluation results in an expression which describes the transformations that have been applied to the variables. An example usage would be the evaluation of a sub--routine with the input parameters as symbols. After the evaluation, the output parameters would be an expression which described their value in relation to the input parameters. Difficulty arises when the evaluation reaches decision statement whose conditions depend upon the value of variables whose values are represented as symbols. The tester is then prompted for the outcome of the decision and this is added to the symbolic state of the program [106, 34, 67].

3.4.2.1.3. Anomaly Analysis

Programs are statically analysed using flow-graphs analysers. The graphs produced can be used to detect anomalies such as unexecutable code, uninitialised variables, unused variables. The analysers are based upon the grammar of the programming language being analysed. Only simple errors can be detected because they do not perform complex reasoning about the state of variables used in programs.

3.4.2.2. Dynamic Techniques

Dynamic techniques involve the execution of the program under test with various sets of test data. Test data is derived from the program's logic, and is designed to exercise a part of the program. The majority of techniques use program coverage to determine when an 'adequate' level of testing has been performed.

3.4.2.2.1. Input Partitioning

Input partitioning in its most general sense is the partitioning of the input parameters with respect to a chosen criteria. The criteria can be a black-box technique such as equivalence partitioning, or a white-box technique which use coverage criteria (such as those described below).

All techniques based on coverage criteria can be considered as input partitioning techniques [53]. Information from the specification, design and source code can be combined to provide a more comprehensive testing technique. This combination of techniques is addressed in section 3.4.3.

3.4.2.2.2. Coverage Techniques

White-box testing techniques are used in conjunction with their appropriate coverage measure to determine when enough test cases have been generated. The coverage measure guides the testing by analysing the segments of the program which have not been exercised. Coverage using simpler coverage measures is easier to achieve than with the more complex ones. However, the simpler the technique, more errors are likely to still exist in the code.

The different coverage techniques are described below, ranging from statement-coverage through to full path coverage.

Statement coverage: this is by far the simplest of all the coverage techniques. For it to be satisfied, every reachable statement in the program must have been executed at least once.

Branch (or decision) coverage: this requires that every decision statement in the program must have been executed with its outcome taking both TRUE and FALSE. This technique provides a slightly greater level of test coverage than statement coverage [106].

Condition coverage: this requires that every condition that makes up every decision statement must be evaluated to both of its possible outcomes (TRUE and FALSE). This provides a greater level of coverage than branch coverage [106].

Multiple condition coverage: all outcomes for all conditions of each decision must be produced. This produces a greater level of coverage of the program [106].

Path coverage: this requires that all paths through a program are traversed. This is only feasible in very small and simple programs. It is not applicable to the majority of programs [106].

Decision-decision path coverage: a **Decision-Decision path** (DD-path) is a sub-path through the program which starts and ends at a decision statement. An objective of DD-path testing is to provide small sets of test cases which in aggregate exercise each DD-path at least once [103]. This technique provides a similar level of coverage to branch coverage.

Domain coverage: **domain testing** represents a specialisation of the general path coverage technique. It does not attempt to test the complete set of all paths through the program. A domain error is a fault in a program whereby a specific input causes execution to follow an incorrect path through the program resulting in an error [66]. The input domain of the function must be linear, not discrete. Also, the technique cannot be used to detect missing paths, or coincidental correctness (input follows and incorrect path, but arrives at the correct answer). It is infeasable for large programs, or programs which have a large number of conditional statements.

LCSAJ Coverage: LCSAJ is an acronym for Linear Code Sequence and Jump. It is a sub-path through a program which begins at either the start of the program, or at a statement which would cause a jump in the program's execution. It is terminated at either the end of the program, or a statement which causes a jump in the execution of the program. A jump is categorised as any statement that causes execution to continue at a statement other than the next one in the source file [155, 59].

3.4.2.2.3. Data-Flow Testing

Data-flow testing uses various criteria to select sub-paths through a procedure depending upon the data-dependencies between points where variables are assigned (definitions) and used (uses) [87, 88].

A number of coverage criteria can be used in conjunction with the basic data-flow analysis information. These are:

1) reach coverage. All paths from a variables definition to its uses must be tested. This however does not guarantee branch coverage of a program.
2) required pairs coverage. This is similar to reach coverage, except that if the use is part of a condition for a decision statement, the decision statement must take both outcomes.
3) elementary data context testing [87]. The elementary data context of a block is the set of all variables used. Input variables to the block are those variables that are used within the block before a new values is assigned to them. Output variables are those that are defined within the block.

3.4.2.2.4. Mutation Analysis

Mutation analysis is essentially a technique for measuring the adequacy of a suite of test cases. A program is seeded with mutants and the suite of test cases are rerun. A mutant is a single mutation of

the original program. The ability of a suite of test cases to detect the mutants is a measure of the adequacy of the test suite [22, 23].

A side effect of mutation analysis is that when errors are found, more test cases are developed. These extra test cases are used to extend the adequacy of the test suite. Hence it can be said that the technique can be used for error detection.

3.4.2.3. Limitations of White-Box Testing

As a minimum, white box testing ensures that all the reachable statements of a program have been executed. More complex white-box techniques provide a more complex coverage of the unit. However, some errors cannot be detected by the use of white-box techniques. Errors that correspond to missing paths through the code are unlikely to be detected because the test cases are derived directly from the source code [47].

The limitations of both black and white-box testing techniques when used alone introduces the requirement for another type of testing technique. This is addressed below.

3.4.3. Grey Box Testing

Both black and white-box testing techniques have limitations. Hence, more than one technique should be used. Black-box testing emphasises the external characteristics of the unit whereas white-box techniques emphasise the internal structure and logic. A combination of both white and a black-box technique should be used; this is known as **grey-box testing**. The key to the success of grey-box testing is the combination of techniques which compliment each other and detect different types of errors [154].

Black-box testing should be used first to validate the external input-output characteristics of the unit. This is then complemented with a white-box technique which is used to test the part of the unit which has yet to be exercised according to the chosen coverage criteria [60].

Grey-box testing is justified by Goodenough and Gerhart [47] who define a formal theory of testing. They note:

1) to detect errors reliably, it is generally necessary to execute a statement under a variety of conditions.
2) some paths through a loop require multiple executions to exercise them fully.

They recommend that input partitions should be derived from the requirements, specification, design and the source code of a program. Hence test cases must represent: every significant branch condition, every potential program termination condition, the partitioning of the domain of each variable, and the partitioning of the input domain described in the specification.

3.5. Integration Testing

Unit testing focuses on testing individual units in isolation. The purpose of integration testing is to validate that units tested in isolation also work when integrated into the main program. Integration tests should at least include tests for:

- interface integrity
- functional validity
- Performance

Integration tests should be developed in conjunction with the system's design. Critical modules should be identified and focused upon [127].

3.5.1. Big Bang Integration Testing

Big-bang integration testing consists of integrating all the modules of a system together at the same time. The system is then tested as a whole. This is not an advisable technique as the tester will find it difficult if not impossible to locate any errors.

3.5.2. Incremental Integration Testing

Instead of integrating all the units together at once, integration testing is more effective if performed incrementally. Units are tested in isolation and are then integrated into the program one at a time. There are two main approaches to incremental integration, top-down and bottom-up. These are described below.

3.5.2.1. Top-Down Integration

Top-down integration testing of a program starts by integrating the units at the top of the program's structure. Units called by the main unit are incrementally integrated with the main one. This process is repeated until all the units in the program have been integrated together. The units can be chosen in one of two ways, depth-first or breadth-first. Depth-first integration is the integration of units down the program's structure, whereas the breadth-first is the integration of units across the program's structure.

Top-down integration requires test stubs to simulate units which are yet to be integrated. These stubs are costly to produce, but allow the main functions of the system to be tested at an early stage. The earlier faults are analysed, the lower the cost to alter the program's design and structure.

Sometimes it is not possible to produce test stubs to simulate units. This is especially true when the units involved manipulate complex data structures. This is solved by using the actual units, however this requires that they exist before testing may proceed.

The major disadvantage of top down is the requirement for stubs and the attendant testing difficulties with them. The advantage is the ability to test the major control-functions of the software early on [106].

3.5.2.2. Bottom-Up Integration

Bottom-up integration starts with the low level units in a program's structure. Units are integrated with the units that call them. This continues upwards until the whole program has been incrementally integrated.

Architectural faults in the program may not be discover until large sections of the program have been implemented. This is because the high functions of the program will not be tested until late in the integration testing phase.

3.5.2.3. Sandwich Integration Testing

Sandwich integration is a combination of top-down and bottom-up integration testing. Top-down integration is used for the top-level control functions of the system and bottom-up is used for the subordinate modules at the bottom of the structure.

3.5.2.4. Thread Integration Testing

Real time systems are difficult to test because of the subtle time-dependencies that exist within a program. Time-dependent defects may only cause the system to fail when all processes have particular states. One method of real-time testing is thread testing. Thread testing is a strategy that follows from process testing. The processing of each external event threads its way through the system processes. Thread testing involves the identification and execution of these threads [127].

3.5.3. Integration Techniques

Leung and White [92] define 3 categories of integration errors:

1) *Missing function error:* given a module A which calls module B; values supplied by A are not part of the domain defined by the interface, hence B can not supply all the required functionality.
2) *Extra function error:* module B provides extra functionality that should not be required by module A, however module A calls the superfluous functionality of B. The calls to the extra functionality are considered an error.
3) *Interface error:* the defined interface between the two components is validated. Either arguments of the wrong type or format are passed, or the arguments are in the wrong order. Also, a mismatch between the domains of actual and formal parameters is a serious problem. This is different from (1), in (1) the error is a shortfall in the specification of module B, whereas here the problem is one of illegal values being passed.

The practical objective during integration testing is to detect errors which would not have been detected in the previous phase (unit testing).

They propose two types of tests:

1) *interface tests:* the aim of these tests are to validate the interface between the two modules under consideration. Static interface tests involve the checking of parameter types and formats. Dynamic interface tests involve ensuring that the input domain of each parameter for calls to a module are not violated.
2) *functional tests:* any functional tests that traverse a call to another module should be isolated. These should be applied to the calling module. Some functional test cases for the called module can be *sensitised* to the calling module. This involves analysing values that can be

passed as inputs to the calling module which cause the required values to be passed to the called module. The sensitising process is a complex and time consuming one.

White-box tests can also be used for integration testing. Techniques such as data-flow testing have been adapted to integration testing [54]. Other techniques include MM-Paths (a module analogy of DD-Paths) [78] and the path reduction technique [52].

3.6. Test Data Adequacy

With a plethora of techniques available, some must be more effective with some sorts of programs than others. Test adequacy criteria have been developed by Weyuker [147, 148] for the comparison of different testing techniques. Each criterion describes a different property that the ideal testing technique should possess.

3.7. Summary

This chapter has outlined software testing within the software lifecycle.

Human or automated test oracles are used to validate the outcome of all test cases. Two activities within the testing phase are unit and integration testing. Unit testing involves validating units in isolation from the surrounding program. Test stubs and drivers are required to simulate the surrounding modules. In contrast, integration testing involves testing units in combination, these units will have been previously unit tested in isolation.

Testing techniques are categorised into two main sections, black-box and white-box. Black-box techniques treat the code under test as a black-box with inputs and outputs hence test case are derived from the specification alone. In contrast, white-box techniques derive test cases from code and the specification.

This chapter has described both types of testing techniques that are applicable at the unit and integration testing level.

Chapter 4.

The Testing of Object-Oriented Programs

4.1. Introduction

The previous chapter outlined the great wealth of traditional testing techniques currently available. However, few of these have been applied to object-oriented programs. This chapter reviews the published literature that is applicable to the testing of object-oriented programs.

The literature survey is split into two main sections. The first, (section 4.2.) discusses the literature that concerned with conceptual and fundamental issues affecting the testing of object oriented programs. This includes discussions on verification and validation, the use of an object's state during validation, and the effects of ordering method invocation.

The second main section, section 4.3., compares and evaluates practical techniques that can be used to test object-oriented programs. Each technique is evaluated against a list of criteria which are outlined at the start of the section.

4.2. Testing Theory

4.2.1. Error Taxonomy

Overbeck [109], reviews the "state-of the art" literature and research into the testing of object-oriented programs, outlining and evaluating the majority of the techniques that are discussed later on in this chapter. From the literature an error taxonomy of object-oriented programs is drawn, based upon earlier rudimentary work by Purchase and Winder [115]. Overbeck also incorporated additional error categories from the work of Trausan-Matu et al. [136].

The error taxonomy is a highly level conceptual view of the problems that can occur, such as misinterpreted specifications, or overlapping functionality between objects. Nine categories are provided for applying to the unit testing of objects. The same nine categories are of a high conceptual level enabling their application to integration testing as well. The description of each category is augmented with examples of the manner in which the errors would manifest themselves.

The categories do not provide any insight into errors that occur within the semantics of the code (a potential list of these is provided in chapter **9.** for the evaluation of state-based testing). The granularity of the taxonomy should be increased to provide testers with an insight into the types of errors they will find. This improvement is also discussed by Overbeck.

4.2.2. Verification and Validation

In the previous chapter, verification, validation and testing (VV & T) were introduced. Their application to the life-cycle for object-oriented programs is discussed by Graham et al. [48], Jacobsen et al. [73], and Trausan-Matu [136]. The object-oriented life-cycle can follow the traditional waterfall model of software development. Of the development phases, neither system nor acceptance testing is required any adaptation for object-oriented programs, because neither require any specific knowledge about the development method or the program's structure. In contrast, both unit and integration testing are affected.

Murphy and Wong [105] also outline the application of VV & T to the life-cycle, however they concentrate upon a more iterative model of life-cycle, such as the fountain model of Henderson-Sellers and Edwards [57]. The main emphasis of Murphy and Wong's discussion is the cluster - a cooperating

set of objects that have a common aim. They provide a model incorporating test planning and design into the cluster development phase.

Thuy [135] concentrates upon the quality aspects of testing object-oriented programs. He categorises the various relationships that can exists between objects within a system. These are used to stipulate a number of "rules of thumb" that can be followed by developers of a system. Use of these rules will enhance the testability of a product.

4.2.3. Allocation of Testing Resources

In addition, Murphy and Wong directly address the testing of an implementation. They provide guidelines for testing both at the class, and the cluster level. Clusters are categorised into two groups depending upon whether they are reusable, or specific to the application under development. All classes within reusable clusters are tested, whereas, for application-specific clusters only classes that meet specified criteria are tested. The only use for this categorisation is to prioritise the clusters that are tested when testing resources are restricted. Although this ensures that clusters reused on other projects have been tested, an improved categorisation would prioritise the classes by criticality. Clusters that are critical to the functionality of the system should have a higher priority than those that are less important regardless of whether they are reusable or not.

4.2.4. The Use of an Object's State in Testing

An object is defined by its state and associated behaviour. The state is of central important to the correct functioning of an object. The state of an object can be viewed in two ways: either conceptually, or physically. Conceptual states are those which are considered significant from an external viewpoint. For example, some conceptual states for an object representing a set of numbers might be: an empty set, a set with the maximum number of elements, a set of odd numbers, and a set of even numbers.

In contrast, physical states are the actual values stored in the attributes of an object. Although physical states can correspond to conceptual states, they are an internal view of the state (a white-box view). The vast majority of techniques that consider state, use the conceptual state rather than the physical one.

Berard [11] notes that traditional testing techniques take little notice an object's state "prior, during, and after execution" of a operation. The state is central to the correct functioning of an object, as they can be considered as a mapping from their input domain to their output domain.

A technique based upon state-transitions for traditional procedural programs is presented by Beizer [9]. The states used are conceptual and are derived from the programs specification alone. The inputs to the program are analysed for values that will cause a state transition within the program. These are then represented as unique input symbols in a model. The outputs generated by the program are analysed in a similar fashion to produce a set of output symbols. The program's states, input symbols and output symbols are combined to form a finite state automaton model of the program under test. The model is then used to generate tests based upon the expected state-transitions.

The technique relies upon the output from the program for the detection of errors. This causes problems when the program has no change between outputs for two different states. In addition, the derivation of the states from the specification is ambiguous and difficult to determine with any accuracy. Different testers will derive different states for the same program. Also, the encoding of output values into unique symbols is rarely an easy task. A white-box technique also based upon finite state machines is presented later in chapter 5. A similar technique for testing finite state automata models within software design is presented by Chow [28].

If the current state of the object is treated as an input to all operations, and an output from all operations, then the state transitions that an object undergoes can be validated. This method is described by Berard [12] and Jacobson et al. [73].

4.2.5. Unit Testing of Objects

The inapplicability of traditional testing techniques has been suggested by Smith and Robson [122]. They state that

> "with object oriented programs, it is not possible to think in terms of conventional static or dynamic testing"

This is qualified when they describe the Von-Neumann traditional model of processing as an active processor with passive data, and the object oriented model as a passive processor with active data. This statement infers that the data within an object can change at any time without requiring the invocation of an operation. This is an incorrect definition, object-orientation is a method for the construction of software, not a method for the execution of software. Although the state of the object

may have changed before the operation is called again, it did not change of its own accord, it changed only when an operation provided by the same object was invoked.

Frankl and Doong [35] claim that testing operations in isolation, as a mappings from an input space to an output space, shifts testing away from the essence of data abstraction - the interaction of operations. This view is based on a purely external view of the class, whereas testing must validate the internal structure and mechanism of a class as well as its external behaviour.

Smith and Robson also discuss a number of other potential problems with the testing of object-oriented programs, however they do not propose any solutions.

Dorman [36] describes two unit testing approaches: unit testing from the bottom of the class hierarchy upwards, and isolation unit testing. Unit testing from the bottom of the class hierarchy upwards involves starting with the class that use no other classes. These are validated first, and then used in the validation of classes further up the inheritance hierarchy. Dorman argues that this technique is only applicable for small projects that do not involve many classes.

The other technique involves surrounding the class under test with test stubs and drivers to facilitate its validation in complete isolation from the surrounding code. This technique involves considerable additional effort in the simulation of a complex and significant portion of the system. A more efficient system would involve those components that have already been validated. Simulations would only be provided for the unvalidated portions of the system, or classes that are difficult to control directly such as external hardware interfaces.

4.2.5.1. Size of Unit

Traditional unit testing is concerned with testing units which are isolated from the surrounding program. The unit is normally the function or the module. However, for object-oriented programs the smallest unit is the class (or object). This is stated by Fiedler [41] and Berard [10, 12], Turner and Robson [138, 142] and Dorman [36]. Dorman, and Turner and Robson conclude that the attributes of a class cannot easily be separated from the operations due to the high level of coupling between the state and most of the operations. This does not mean that operations cannot be validated in isolation from each other, it means that the attributes of the classes must be present and accessible.

4.2.5.2. Method Ordering

Berard proposes that confidence is gained in operations that report the current state of an object (without altering it). These can then be used to validate other operations. This concept is extended by Turner and Robson [141] who provide a categorisation of operations and suggestions for their order within the testing. The ordering reuses previously validated operations in the validation of others. This is essentially a black-box technique where an object's operations are used to create and manipulate an object. It is a highly iterative process, requiring extensive and careful planning to avoid using any unvalidated operations in the testing of another. It is unlikely that complete coverage of all routines can be achieved without the need for operations which have yet to be exercised. This means that the tester must ensure that the only previously validated paths through operations are used in the validation of others. This is a difficult task which may not be cost effective.

Dorman [36], on the contrary proposes that operations should be tested in isolation, as this reduces the dependency upon unvalidated code. Test case scenarios are generated directly using test drivers. This requires more code to be written, however each test case is then isolated from the remainder of the class.

4.2.5.3. Test Adequacy

Fiedler [41] (discussed below in section 4.3.1.4.) states that the object-oriented paradigm suggests only black-box testing is required because of the focus on the external view of an object. However, he admits a more robust testing method including white-box testing is actually required. This point is echoed by Perry and Kaiser [112] who apply Weyuker's test data adequacy criteria to object-oriented programming.

Weyuker's criteria (see [147] were originally designed for the procedural style of programming, and are used to draw comparisons between different testing techniques. Of the criteria, seven are reported as being "intuitively obvious" and/or have no special effect on object-oriented programming. Of the remaining criteria, two affect unit testing directly, they are the antiextensionality and the general multiple change criterion.

The Antiextensionality criterion states that if two functions compute the same function, then a test suite adequate for one is not necessarily adequate for the other. The criteria that determines if testing is complete must be dependent upon the programs structure.

The General Multiple Change criterion states that if two units are syntactically similar, a test set adequate for one unit is not necessarily adequate for the other. This infers that the testing technique is also dependent upon the specification or functionality of the unit, and not solely the program's structure. Although two programs have a similar structure, they may have different specifications (and functionality) requiring different test suites. Together, the Antiextensionality and the General Multiple Change criterion require both black and white-box testing to adequately validate classes.

4.2.5.4. Inheritance

Unit testing has so far considered the class as a collection of attributes and operations, however, classes defined by inheritance introduce the potential for more problems. This is discussed by Perry and Kaiser [112] when they consider the affect of the Antidecomposition criterion and the Antiextensionality criterion (described earlier) on the testing of child classes.

The Antidecomposition criterion states that a unit is likely to require retesting if it is taken out of the context of the surrounding. The context of an operation is defined by the physical states that can be generated by the other operations of the class, these states are used as input to the operation in question. If the physical states generated by the operations of a child class are not a subset of the parent's states, then the context of the operation has changed and it must be retested. This is discussed in detail by Turner and Robson [139] and can be found in section 6.4.

A child class can only change the context of an operation if operations of the child class have access to the attributes of the parent class. If they do not have access, they are unable to generate states which would be out of the context of the inherited operation. Another potential problem is that the operations of the child class are unable to generate enough of the states required by an inherited operation to be able to perform its functionality correctly. Hence inherited operations require retesting in child classes, although they are unlikely to require completely retesting.

The antiextensionality criterion (as discussed earlier) implies that two similarly specified functions are likely to require different structural test suites. This has implications for the overriding of operations in a child class. Although. the new version of the operation will have a similar specification to that of the overridden one, it is likely to requiring new white-box test cases as the implementation will have changed.

There are two main techniques for testing classes defined by inheritance: analyse and retest only those operations that require retesting, or retest all inherited features regardless of previous testing.

Fielder [41] expresses the popular view that when testing child classes, inherited operations only require minimal testing. Cheatham and Mellinger [27] echo Fielder's view by stating that with a Smalltalk library of tested classes, 40-70% of unit testing is "free". However, no details of a study are provided to substantiate this claim. A much more comprehensive and justifiable algorithm is presented Harrold et al. [55].

The alternative approach has been proposed Murphy and Wong [105] and Hoffman and Strooper [63, 131]. Murphy and Wong describe a mechanism for allowing child classes either to inherit its parents' test cases for an operation, or to override them. Hoffman and Strooper rerun all an operations test cases from its parent class. They justify this by making the testing process automatic and efficient. All of these techniques are discussed in more detail in section 4.3.2.

4.2.6. Integration Testing of Objects

Graham et al. [48] discuss the application of integration testing at two levels: the operation, and the class. The integration testing of operations within a class is defined as **Intra-class** testing. Whereas **Inter-class** testing is defined as the testing between classes. Intra-class testing involves ensuring that operations communicate correctly via the state. Inter-class testing involves validating that objects not only call the correct operations in other objects, but that the correct parameter objects are passed.

Both intra-class and inter-class testing are included in the incremental algorithm of Harrold et al. [55]. Harrold et al. consider inter-class testing to be an analogy of intra-class testing, but with operations communicating between separate classes. However, inter-class testing has a different focus from intra-class testing. Inter-class testing is similar to traditional integration testing which concentrates upon the interface between two units. Inter-class testing must also concentrate on the interface between two classes and correctly manipulate and validate the states of multiple objects. Harrold et al.'s algorithm is discussed in section 4.3.2.

Dorman [36] discusses the integration testing of classes, advocating a 'big-bang' approach for small projects, and a gradual, incremental approach for larger projects. No indication on the size of a small project is given. The 'big-bang' integration method is not recommended, no matter how small the project is due to the increased complexity of exercising, tracing and debugging the paths through the code.

4.2.7. Specification of Test Cases

A generic specification technique for test cases is detailed by Berard [11]. A generic technique is advantageous when considering the automatic execution and validation of test cases. Each test case consists of a number of sections. The majority of the sections are the same for programs written in OOPLs, as they are for programs written in a more traditional way. However, in addition the generic technique requires an unambiguous specification of the starting and finishing states of each object involved. If any of the intermediate states involved in the transition from the starting state to the finishing state are of interest then they must also be specified. Berard advocates the use of conceptual object states for this purpose, although physical states can be specified just as easily.

Both Murphy and Wong [105] and Frankl and Doong [35] also describe the content of their test case, however their technique although as formal, does not include any information that is not required for the automatic execution of the test cases. Both of these types of test cases consist of sequences of calls to operations of an object under test. However neither technique requires any specification of an object's state as it is not needed for the technique.

Murphy and Wong's, and Frankl and Doong's techniques are detailed in section 4.3.1.

4.3. Testing Techniques For Object-Oriented Programs

This section discusses techniques that can be used for validating object-oriented programs. Each of the techniques will be evaluated against as many of the following criteria as are appropriate:

1) Is the technique a black-box, white-box, or mathematical technique (such as program proving) ?
2) Is the object's state taken into account during testing ? If so, does the technique use the object's conceptual states, or the physical states ?
3) Can an automated oracle be generated for the technique, or has one been produced ? If one is available, to what degree does it restrict the types of test cases used ? Is the oracle fully automated, or does it require partial interaction with the tester ?
4) Can operations be tested in isolation from each other ? Can sequences of operations be validated ?
5) Does the technique have any special requirements before it can be applied to a class ? Does it require an particular expertise (such as mathematics for formal proofs) ? Does the technique require alterations to the class under test ?

6) Is the technique systematic ? Is it clearly defined so that no matter who applies it, it always results in the same set of test cases ? Or, does it rely heavily upon the tester's experience ? Is the technique easily automated ? Can the test cases be automatically generated using a suite of tools ?

7) Is the technique applicable to all classes ? What restrictions does the technique place upon the classes it can test ?

8) Is the technique applicable to integration testing ? Is it designed solely for unit testing ? Can the technique handle complex interactions between objects ? Can the technique (or the tools) handle the creation of complex objects that are passed as parameters to the class under test ?

9) Does the technique have special facilities for handling child classes ? Does it include a strategy for detecting the change in context that can occur within inherited features ?

10) Does the technique generate large or small numbers of test cases ? Does the technique require the generation of large numbers of test cases to achieve a reasonable coverage of the class ? Are the test cases relatively easy to generate ? Large numbers of simple and easily executed test cases must be contrasted with small numbers of complex and difficult to generate and execute test cases.

11) What is the main focus of the test cases ? Does the technique concentrate upon paths throughout the code, the external behaviour of the object, or the state transitions of the object ?

Only those criteria that are relevant to each technique will be discussed. In addition, some criteria cannot be applied to some techniques as it has been impossible to obtain the enough information.

4.3.1. Unit Testing Techniques

4.3.1.1. Formal Approaches With ADTs

Abstract Data Types (ADTs) are a specification technique for modelling software components using mathematical expressions. Mathematical proofs can be used to verify that an implementation of a software component conforms to its specification. Olthoff [108] is the first known application of this technique directly to the validation of object-oriented programs, although early work on the proving of ADTs is attributed to Hoare [61] and Guttag et al. [50].

Olthoff describes an approach using a modified version of Pascal (ModPascal) that includes constructs for defining classes, a method for extending other objects (known as enrichment) and for the inclusion

of information connected with an ADT's specification. The paper focuses on the axiomatic specification[9] method which is supported by the ModPascal system.

Formal specifications require an experienced mathematician to be able to manipulate them with any degree of competence and success. At present very few tools and techniques are available to aid the tester in either the production of specifications, their validation [76, 113], or the verification of an implementation.

An earlier approach using formal specifications is described by Schorre in [119]. He outlines a program verifier which is used for proving that an implementation meets its specification. A considerable competence in mathematics is required to express the mapping from the specification to the implementation. The user of the tool specifies operations, including loops, that are undertaken by a symbolic evaluator to demonstrate the conformance of the implementation. Human errors may be introduced into either the specification or the mapping from the specification to the implementation reducing the accuracy and validity of any proof.

Both of these formal approaches are heavily dependent upon the completeness of the ADT's specification. The majority of the proof that the implementation is a valid representation of the specification is left to the mathematician. If an axiomatic specification is used, operations are verified in unison with the rest of the ADT, whereas algebraic specification allows individual operations to be verified in isolation. Mathematical proofs can be applied to whole programs as well as individual units, although they are large, time consuming and cumbersome, and therefore not practical to apply to large projects.

4.3.1.2. Informal Approaches With ADTs

Rather than using the specifications for mathematical proofs, another approach is to derive test cases from the specification for validating the ADT. This technique is simply to apply and requires less mathematical expertise, although it can never prove validity it can only demonstrate it. A number of different approaches are described below.

Gannon et al. [44] describe their Data-Abstraction Implementation, Specification and Testing System (DAISTS). DAISTS is a system which aids a user in the implementation of an ADT and then

---

9 An axiomatic specification describes an ADT in terms of the interaction between the operations provided. This is in contrast to an algorithmic specification which describe each operation's functionality with an algorithm.

augments it with axioms from its specification. Test cases are applied to the ADT to test the consistency of the implementation by evaluating the axioms at strategic points in the programs execution. The axioms are converted into code that can be executed by DAISTS.

Results are collected by "execution-monitoring." Example results would be: data for which an axiom failed, any statements that were left unexecuted, and any expressions whose value remained constant for every evaluation. Gannon et al. note situations when errors in the implementation would not be detected by the test cases designed to exercise the axioms. Consequently, the system also includes a structural coverage criterion and a facility to detect constant expressions.

The technique incorporates both a black and a white-box approach to validation, although no direct use is made of information about the ADT's state. The test cases focus on exercising the axioms which form the main oracle of the system.

The technique is not applicable to objects that cannot be describes by an axiomatic specification. In addition, no method or guidelines are provided which describe the generation of test data for exercising the axioms. This point has been addressed by both Jalotte and, Frankl and Doong.

Jalote [74] describes an automated oracle that uses an axiomatic specification to determine the outcome of each test case. The use of an automated oracle reduces the possibility of human error and automates an arduous task. Jalote also describes a test-point generator which aids the tester in the generation of test cases by analysing the specification for boundary conditions. As mentioned earlier, not all objects can easily be modelled as ADTs, so, the requirement for axiomatic specifications restricts the application of the technique.

Axiomatic specifications consist of two parts: the syntax part defines the parameter and return value types for the operations of an ADT, and the semantic part which describes the functionality of the operations. Jalote and Caballero [75] extend Jalote's earlier work by describing a tool for deriving test cases directly from the syntax section of a formal specification. The test cases are simple combinations of operations which exercise the ADT's implementation on a basic level.

The test cases generated by this extension are not based upon the functionality of the class as the semantic part of the specification part is not used. The test cases will provide only a low level of testing for the ADT.

Both Jalote's earlier approach and the later extension are purely specification based approaches and should be used in conjunction with a white-box technique, although this aspect is not discussed by Jalote and Cabalero.

Although the techniques require a formal specification, they are fully automated in the generation, execution and validation of test cases. The automated techniques require little intervention on the part of the tester. Although extra test cases will also be required to validate those aspects of the ADT which cannot be expressed within an axiomatic specification.

The technique is likely to be less effective than DAISTS as no coverage criteria is used and there is no detection of constant expressions, although a much greater part of this technique is automated than with DAISTS.

Another technique using axiomatic specifications to generate test cases is presented by Frankl and Doong [42]. Their technique and its associated set of tools (known as ASTOOT) use the formal specification to generate sequences of calls to operations. Each test case consists of two of these sequences and a tag. Both sequences are applied to separate objects and the two resulting objects are then compared. The tag in the test case is used to determine if this comparison should find the two objects observationally the same, or not.

A tester supplied routine EQN forms an integral part of the automated oracle. EQN performs the comparison between the two objects. It can compared either the conceptual, or the physical states of the two objects, although Frankl and Doong suggest conceptual comparisons as the technique is uses an external view of the objects.

A major disadvantage with this technique is that an object's state is only compared with another object, it is not determined absolutely. Hence, if both objects were in the same wrong state, the error would not be detected by that test case. Direct validation of an object's state cannot be performed by a black-box technique.

ASTOOT consists of three components; the driver generator, the compiler and the simplifier. The compiler and the simplifier combine to form an interactive test generation tool aiding the tester in the generation of test cases. The simplifier uses rewrite rules provided by the compiler to produce another sequence of operations, together they form a test case. The driver generator allows the execution of automatically or manually generated test cases.

If an ADT T_d is expressed as a descendant[10] of T, then it inherits all of T's operations. The operations of T_d are constrained by the axioms of T and can be constrained further if required. Nevertheless, not all test cases for the parent T are rerun for T_d, only the test cases for redefined operations are rerun.

---

10 A descendant ADT is similar to a child class.

This strategy is founded in black-box testing and hence it does not satisfy the antiextensionality criteria discussed by Perry and Kaiser.

Frankl and Doong also introduce an exception to the strategy for testing T_d, if the child class is simply expressing code-reuse then it does not need to satisfy the axioms of its parents. It must therefore require complete retesting in its new context, although this is not discussed. Frankl and Doong do not address the potential change of context that can occur when an operation is inherited. Therefore the addition of this exception to the algorithm is dangerous.

Using Frankl and Doong method, operations cannot be validated in isolation as the technique relies upon sequences of operations. This restriction is inherent in the use of axiomatic specifications which describe an ADT . Operations are not described in isolation, instead, the functionality of an ADT is expressed in terms of the relationships between the operations. As with the other techniques discussed so far, the restrictions caused by the use of ADT specifications also apply to this technique.

The technique itself is applicable to integration testing of objects, although at present, the tool set does not seem to support the construction of complex parameter objects restricting the technique's application to objects which requires only simple parameter types, for example, numbers or strings. Large numbers of test cases are generated for simple ADTs such as stacks and trees (2000 upwards). The generation of test cases is not fully automatic and so will be a long and potentially laborious process. Furthermore, the method cannot be performed exhaustively as it is reliant on sequences of operations, which can be potentially infinite in length.

Frankl and Doong contrast their method with DAISTS, stating that their method is likely to discover more errors because test cases' can result in either equivalent states, or dissimilar states. However, their method does not include any form of structural coverage criteria, nor does it include constant expression detection.

As a follow-up to their earlier article, Frankl and Doong [35] present an assessment of their technique. They outline an experiment conducted with two example ADTs both seeded with a "purposeful" error. During the experiment, a number of questions were asked such as "How does the length of the sequences used by test cases, affect the detection of errors ?" Their results draw the conclusion that larger parameter ranges with longer sequences of operations seemed to be more effective than shorter parameter ranges with shorter sequences.

4.3.1.3. Reasoning About ADTs

In contrast with the class level testing of Jalote and, Frankl and Doong and the verification of Olthoff, Leavens [89] describes a method for reasoning about the conceptual relationships between ADTs using algorithmic specifications. However, he does not discuss in detail the testing of an ADT's implementation. His method is more applicable to the validation of the design.

There is no direct correlation between the ADT's used for the reasoning and the classes used for the implementation. This contrasts sharply with the other techniques described above which use ADTs to model classes directly, providing a direct correlation between the specification and the implementation. A direct correlation aids the tester in the development of test data for use within test cases.

Leavens suggests that an implementation can be verified by using abstract relations that map abstract values of an ADT to concrete physical values. This is a mapping of conceptual states to the physical states that will be used within the object.

4.3.1.4. McCabe Cyclomatic Testing Method

Fiedler [41] describes the application of the McCabe cyclomatic testing method [97] to an object-oriented project written in C++. The McCabe method is a traditional testing technique for the analysis and testing of independent paths through an application. Although object-oriented programming emphasises the external behaviour of objects, Fiedler states that testing must consist of both black and white-box testing including some form of path coverage.

Results from the test runs were verified by a simple file comparison with the expected results. This technique is error prone, expensive and cumbersome because of the time required to generate the expected output exactly. Nevertheless, it is an easy technique to use when contrasted with the development of complex self-verifying test cases or automated oracles.

The McCabe technique does not directly address the use of an object's state for testing, although Fiedler does advocate the application of a type of equivalence partitioning to the object's conceptual states for the generation of the black-box test cases.

Neither the black, nor the white-box testing techniques proposed by Fielder can be used to test operations in isolation directly, nor can the test cases be automatically generated. All classes can be tested with this combination of techniques, including those that are defined using inheritance. Fiedler

proposes only minimal retesting for inherited operations rather than the maximalistic approach of Murphy and Wong described below. He does not directly address the potential change of context that can occur with inherited operations.

Berard [12] and McCabe [98] also describes the application of the McCabe technique to object-oriented programs in a similar manner to that of Fiedler.

4.3.1.5. Module Test Case Generation

Another black-box approach is described by Murphy and Wong [105], who use extensions to a tool (PGMGEN) from earlier work by Hoffman into test case generation for C modules. Hoffman's early work [62, 65] into the practical testing of modules is very similar to that of Gannon et al. in the DAISTS system (see section 4.3.1.1. above), although, Hoffman uses test cases derived from informal specifications. Test cases consist of sequences (referred to as traces) of calls to operations exported by a module (analogous to the operations of a class). The result of each test case is obtained by evaluating an expression contained within the test case, which is then compared with the expected value.

This work was later extended by Hoffman and Strooper [63, 131, 64] to include the use of *testgraphs* and prolog programs for defining the traces applied to the module under test. A *testgraph* is a directed graph of nodes and arcs used for expressing conceptual state transitions of an object. The nodes represent potential states the module can be in. The arcs between the nodes represent sequences of calls to the module's interface that *should* cause the transition between the two nodes.

All arcs and nodes are augmented with 'labels'. The 'labels' on the nodes contain the interface calls to validate that the module is indeed in the expected state, whereas the labels on the arcs contain the interface calls which should cause the transitions from one node to the next. Hoffman and Strooper state that the sequence of calls augmenting each node must only contain calls that do not alter the state of the module. Correspondingly, the sequence of calls augmenting each arc must only contain calls that alter the state.

The states chosen for the nodes of the *testgraph* are conceptual ones, not physical ones. They are chosen using an interval technique similar to a combination of equivalence partitioning and boundary analysis. However, these black-box techniques are difficult to apply to a module's conceptual states. In addition, the results of applying the interval technique will be highly subjective and dependent upon the tester involved.

States are validated by using the interface of the module, direct access to the attributes is not used. If an error is present both in the operations that set the state, and the operations that read the state, it is conceivable that the error will go undetected. The technique suffers from the same disadvantages as black-box techniques in general, that is, an inability to ensure that all the code has been exercised.

If combined with an appropriate white-box technique, this technique would be more effective than the DAISTS system of Gannon et al. (discussed above). The provision of an automated oracle for test case validation and the ability to automatically generate test cases from the testgraph makes the technique more efficient in it use of testing resources.

Hoffman and Strooper [64] address the testing of child classes by rerunning all the test cases from a parent class. They hope to achieve this by making the execution of test cases inexpensive and efficient. However, as class hierarchies grow in depth, the number of inherited operations grows exponentially. This leads to a greater level of testing resources being used for rerunning old test cases. The increase in resources required must be compared with the effort required to analyse which test cases need not be rerun (Harrold et al., see section 4.3.2.).

4.3.1.6. Black-Box Techniques

Traditional black-box techniques can be used to partition the input space of a unit to reduce the potentially infinite ranges to more manageable ones. The techniques can also be applied to object-oriented programs. Berard [12] outlines a number of different techniques, including equivalence partitioning, boundary analysis and cause-effect graphing. Each of these are applied to the conceptual state of an object, producing test cases which are independent of the object's implementation but dependent upon its specification.

Jacobson et al. [73] also advocates the application of equivalence partitioning to the conceptual states of an object. The resulting partitions are combined with a list of stimuli to form a state matrix. The stimuli are analysed in a similar manner to the state partitions, each stimulus is a call to an operation with parameter values that considered to be significant. The state matrix is then used to predict the effect of a sequence of operation calls. The matrix is used in a similar manner to the testgraphs of Hoffman and Strooper, however the matrix is not automated in any way. This technique is also a purely black-box approach.

All validation is performed through the provided interface, hence, operations cannot easily be tested in isolation.

He also advocates the use of white-box techniques such as decision decision-path coverage. When used in combination with the black-box techniques above, a program should receive adequate testing. However, none of the techniques described by Jacobson et al. are easily automated. This increases the possibility of human-errors introduced by the tester, and increases the resources required to perform adequate testing.

4.3.2. Inheritance

Harrold et al [55] describe an algorithm for reducing the number of test cases which need to be rerun when testing a child class. It is contrasted against the other method suggested by Hoffman and Strooper, and Murphy and Wong - the flattening the class structure where each class is tested as if it provided all of the operations. The flattening method causes a combinatorial explosion in the number operations that must be tested as the size of class hierarchies increase, whereas the reducing algorithm requires substantial effort during the analysis.

The suggested order of testing for an inheritance hierarchy of classes is (intuitively) from the top downwards, that is, starting with the parent classes and continuing down the hierarchy. Harrold et al. propose testing each operation in isolation, followed by the interactions between operations of the same class.

State-based testing which is to be described later on in this thesis (section 5.) allows the interactions between the operations to be tested in isolation. Its potential incorporation into Harrold et al.'s algorithm is described in section 6.4.

The algorithm uses test histories from parent classes. A test history associates a test case with the operations it tests. Test histories for child classes are created by incrementally updating the test histories from their parent classes with information about the child classes' differences from the parents. From the new test history, test cases from the parent class can be identified for reuse along with any operations and attributes for which new test cases must be generated. Six categories of operations are introduced, covering all possible ways of defining a operation in a child class. These form the focus of the algorithm.

Inheritance is remodelled in terms of a class and a modifier, when the modifier is applied to the parent class it produces the child class. The modifier also forms part of the incremental update that is applied to the parent's test histories. This allows the inheritance hierarchy to be decomposed into a set of modifiers.

Very little information is provided for the actual generation of the test cases. Nevertheless, the paper states that both black and white-box test suites should be produced for both the isolation and the integration testing of operations, the work of Perry and Kaiser [112] is used to justify this. Also, no study is presented showing that classes are still adequately tested after the application of the algorithm to the test suites.

Murphy and Wong [105] propose a different method for the testing of child classes. Their algorithm reruns all test cases from the parent class unless they have been overridden. Each test case is given a unique identifier. Two test cases are permitted to have the same identifier only if one test case is from a parent class. and the other is from the current class (a child class). Overriding of test cases therefore uses a mechanism similar to the one used for the overriding of operations. All test cases are automatically generated, compiled and executed by the tool. This causes a combinatorial explosion in the number of test cases as the depth of the inheritance hierarchy increases.

Hoffman and Strooper [64] also propose this technique for testing child classes.

Thuy [135] also notes the problem of testing classes that have been defined by inheritance. He addresses a number of points including: a reduction in the number of methods of a parent class that require retesting when testing them in the context of the child class, and the testing of abstract base classes and generic classes. He presents by way of examples an informal algorithm for reducing the testing required for child classes that is similar to the algorithm of Harrold et al. (discussed above).

When testing abstract base classes he considers the interaction of operations with deferred operations. From these connections, he briefly outlines the testing of a class by careful analysis of the child classes that provide an implementation for the deferred operations.

4.3.3. Integration Testing

Winfrey [149] discusses a problem of integration testing objects where the source code for some of the classes are not available. For this he suggest replacing them with a 'wrapper' which logs all parameters that were supplied. The 'wrapper' then calls the operation from the original object, logs any return values and returns them to the caller. This technique is discussed with reference to 'two mutually suspicious parties' which refuse to supply source code to each other.

The above cannot be achieved in some languages as it is not always possible to simply rename a compiled object by renaming the file it is in. It has other benefits though, as it can be used for the

integration testing of objects where the source code is available. The 'wrapper' mechanism allows the tester to easily study the objects passed around the system.

4.3.4. Frameworks for Testing Object-Oriented Programs

A number of tools and frameworks have been developed for the testing of object-oriented programs. The majority of these have already been described with the exception of the framework by Smith and Robson [123]. Their framework allows the tester to include specific strategies for testing particular classes.

They describe a number of strategies which may be "slotted" into the framework. These range from simple ones such as the tester guided strategy where a tester suggest combinations of routines to exercise, to more complex ones for applying data-flow analysis. However, no implementational description of the framework is provided. The paper suggests that the strategies can be extended in an object-oriented manner by the use of inheritance, allowing specialisation for a particular application to occur.

The paper concludes that there is a need for the development of new strategies as these form the core of the framework. Also further evaluation is required to determine the "usefulness" of the strategies.

A major omission from the paper is an evaluation both of the framework as a whole, and of each strategy. The framework has the potential for the inclusion of any of the techniques outlined in this chapter, however the difficulty in creating and managing test strategies has not been addressed.

4.3.4.1. Test Case Generation with Cantata

Dorman [36] outlines a testing approach that is supported by the tool Cantata. Cantata is a test generation and execution tool available from IPL Ltd. It supports the creation of test cases from a test script which describes the process and operations to be executed. Also, test stubs can be described which allow the simulation of classes and operations that are not present or available at the time of testing.

The tool provides facilities that enable test case output so that the current state of a test case can be reported during its execution. In addition, Cantata can calculate various coverage measures and metrics for the test cases and the code under test. This includes the statement and branch coverage achieved with the test cases, and McCabe and Halstead's metrics for the code under test.

Cantata has been designed to work with the language C++. It uses the friend feature provided by the language to circumvent the access restrictions when testing private and protected operations. Dorman [36] argues that this approach is more beneficial than directly altering the access privileges which changes the context of the class, however no justification for this statement is provided. Not all object-oriented languages provide a feature similar to the friend feature of C++. In these situations, changes to the access privileges must be performed to enable the test cases to access the hidden operations.

Each class is required to provide a test operation to facilitate testing. Test scripts are used to control the testing including the invocation of operations. The test script describes the test cases that must be performed on the class. All test cases should be isolated from each other, hence any changes are not rippled throughout the test suite.

Cantata also has the facility for calculating the statement, branch and condition coverage of the code under test when the test suite is applied.

4.4. Summary of Chapters 2 to 4

Chapter 2 outlined the basic concepts that are collectively known as object-oriented programming. The technique that will be described in chapter 5 is applicable to a large number of these concepts. It can be applied to the three main phases of an object's lifecycle, namely: its creation, its execution (the invocation of its methods), and its destruction.

Parallel object-oriented programming languages provide difficulties that will not be addressed within this thesis, therefore the technique that will be described is only applicable to object-oriented programming languages (OOPLs) that follow the serial execution model whereby each instruction is executed after the previous one has terminated. Of these OOPLs, those that use dynamic inheritance are also excluded. The ability of a class to dynamically change its parent class during execution means that the extent of an objects state is not static and therefore cannot be currently addressed by the technique in chapter 5. The technique can be applied to those OOPLs known as prototypical OOPLs, although those that use delegation cannot be tested because of delegations decentralising nature.

Chapter 3 introduced the fundamental concepts and outlined many of the traditional techniques used for testing software. All techniques require some method of verifying test results. The entity that validates the outcome of testcases is known as an oracle. Oracles can be either automated, or manual. Manual oracles are labour intensive and prone to errors themselves, therefore automated oracles are becoming an increasing necessity to make testing practical in the large scale.

Of the four levels of testing described (unit, integration, system and acceptance testing), only unit and integration testing are covered any further in this thesis. Neither system, nor acceptance testing is affected by the design and implementation technique and therefore are the same for object-oriented programs as they are for more traditionally implemented ones.

In chapter 3 testing techniques were further categorised into black and white-box techniques. Black-box techniques derive their test cases solely from the components requirements without any regard for the components implementation. In contrast, white-box techniques use the structure and implementation details of a component to derive its test cases. Black-box techniques are not guaranteed to exercise every statement in the component, whereas White-box techniques are not guaranteed to find missing paths within the program because their guidance is drawn from the structure of the code under test.

This chapter (chapter 4) has describe the literature that is concerned with the testing of object-oriented programs. All the techniques describes have been evaluated against a set of criteria and compared against each other where possible.

The majority of the literature agrees that the smallest unit of testing is the class rather than the operation. This has the effect of complicating the testing process. Operations can rarely be tested alone, increasing the difficulty in their testing.

The state of an object is of central importance to its correct functioning because operations use the state to store situations and events that are referenced by the other operations. Ensuring that the object is in the correct state is therefore essential to the objects correct functioning. Of the techniques that address the state of an object during testing, all of them use the conceptual state, rather than the physical state.

A major problem that has not been directly addressed by any of the techniques outlined in this chapter is the creation of complex objects. Most of the techniques described above have only been applied either to simple classes, or classes that are near the bottom of the instantiation tree, hence they require simulation of few others.

The constructor operations provided by an object usually require parameter values. Each of these parameters is itself an object also requiring creation. For a complex object, the chain of objects that must be created and manipulated to create the initial scenario for a test case can become extensive, resulting in a difficult task. This is not easily automated.

These points will be addressed below.

4.5. The Requirement for a New Technique

The summary (above) noted a number of shortcomings in the current techniques for testing object-oriented programs, these are used below to demonstrate a requirement for a complementary technique such as state-based testing.

The vast majority of testing techniques currently available are adaptations of more traditional approaches. Few of these take into account any of the facilities that are particular to object-oriented programming, hence they will be less beneficial. Hence there is a requirement for a technique that directly addresses fundamental aspects of object-oriented programming that is not currently addressed by traditional techniques.

In addition, the technique must be methodical in the generation of test cases, and have an easily determined point when to cease the generation of test cases. This point may arise because all the possible test cases have been generated; this is included as a basic requirement so that the user of the technique has a measurable point when they are to cease testing using state-based testing. This improves the measurability of testing and allows resources to be devoted to other techniques that compliment state-based testing.

An object's state is of central importance to its correct functioning; operations interact and communicate with each other via the state, notifying each other of events and values. Of the techniques reviewed that address the state of an object during testing, presently all of them consider the state conceptually, rather than physically. Hence there is a requirement for a technique that considers the physical values used within the state of an object.

Rather than use a conceptual view of an object's state which can be both difficult to analyse and difficult to map onto the values stored in the attributes of an object, state-based testing uses a physical view of an object states. Physical states are simple to obtain from the design information, in addition to being simple to validate.

If a technique can be automated, it has an improved likelihood of being applied in the real world because of a reduction in the cost of its application. With state-based testing, a semi-automatic oracle can be generated for any class. The tester must chose the transitions that will take place during each test case, the validation of the initial and final states is then automatically achieved with the aid of tools (see chapter 7.). There is therefore a requirement for a technique that models the functionality of a class in such a way that the testing can be practically automated and validated.

Testing of object-oriented programs uses the class as the smallest unit of testing causing a complication of the testing process when compared with traditional unit testing. However, this does not imply that the *whole* class is required for each test case, on the contrary, it only requires that the *attributes* are present along with the current operation under test. State-based testing facilitates the testing of operations in isolation by validating the states involved in the test case directly. If the initial state of a test case is also created directly, the only operation required to be present in a test case is the one under test.

Finally, the operations provided by an object for its creation and initialisation usually requires parameter values. Each of these parameters is itself an object also requiring creation. For a complex object under test, this chain of objects requiring creation and manipulation for the construction of the initial scenario for a test case can become an extensive and difficult task. This is not easily automated. The current version of the prototype tools (see chapter 7.) avoid this problem by requiring the tester to supply a short piece of code that creates the parameter objects required for a test case. It is far easier for the tester to write this short piece of code than it would be for him to instruct a tool to generate the required code, even though it is more human intensive.

For any technique to be useful and practical, it must be effective locating errors within the code under test. The technique must therefore also be effective in achieving execution coverage of the code. This must be done with an optimal, or near optimal suite of test cases.

In summary, the main requirements are:

- the technique must address fundamental conecpts of object-oriented programming,
- it must consider the state of an object during testing,
- it must be possible to test operations in isolation from each other,
- it must be methodical to apply,
- it must have an easily determined end point when no more test cases can be generated,
- it must be effective in locating errors with a near optimal suite of test cases,
- it should use physical values that are used within the objects rather than conceptual ones,
- it must be possible to automate the application of some if not all of the technique.

The remainder of this thesis describes state-based testing for the validation of object-oriented programs based upon the physical state transitions that take place within an object state. An evaluation of the technique is presented to demonstrate the effectiveness of the technique for locating errors within a class. The points raised above are also discussed in the conclusions (chapter 10.) to evaluate whether the technique described in the intervening chapters satisfies these requirements.

Chapter 5.

A New Technique - State-Based Testing

5.1. Introduction

State-based testing is a new technique for the validation of object-oriented programs. It is based upon the modelling of classes as finite state automata. The automata are used to generate the test cases. Each test cases involves the creation of a specific state for the object, followed by the invocation of an operation, and concluded by the validation of the final state achieved by the object.

This chapter describes in detail both the theory and the practice of generating a state-based test suite. It is divided into six main sections (in addition to this one).

The first section (5.2.) describes the terms and concepts that are required to understand the theory presented in the remaining sections of this chapter. Sub-sections 5.2.1. and 5.2.2. introduce the basics of set theory and functions that are needed. Sequential machines, automata, transition tables and transition functions are described in the subsequent sub-sections. Sub-section 5.2.4. provides a concise summary of the symbols that will be used. It is provided as a concise reference allowing the reader to skip the preceding sub-sections.

Automata are used to model the internal representation of classes in section 5.3. A mapping from the physical (classes) to the theoretical (automata) is presented. This involves the creation of a state transition table for all operations. An example is provided for clarity. The special cases of constructors and destructors are also discussed.

Automata are not always readily available in the form of a design, hence an algorithm is needed which can generate one. This is presented in section 5.4. The next section (5.5.) describes the fundamental process involved in generating a state-based test suite from the automata description. It is presented in the form of an algorithm. It is then extended in section 5.6. to facilitate the testing of classes which employ dynamic data-structures as well as static ones. This is done with the aid of a more complex example - a linked list class.

The final section (5.8.) summarises the technique that is presented in this chapter.

5.2. Prerequisite Terms and Concepts

5.2.1. Set Theory

The basic set notation described below will be used throughout the remainder of this thesis to formally and concisely express the process of state-based testing. The information is taken from [116].

A set (denoted by a capital letter) is an unordered collection of unique elements. It can be explicitly specified by enclosing its members in braces {}. For example, a set S,

$S = \{a,b,c\}$

The empty set is represented by the symbol ∅.

$\varnothing = \{\}$

A similar concept is that of the empty element (or string) which is represented by the symbol ε.

A set can have either a finite or infinite number of elements. The number of elements or *cardinality* of a set S is denoted by #(S), where $0 \leq \#(S) \leq \infty$.

Sets can be combined using operations such as union ('$\cup$'), intersection ('$\cap$') and Cartesian product ('$\times$'). However, union and intersection are basic set operations and therefore will not be described here. A Cartesian product is the combination of each member of the first set with each member of the second set. For example, if

$A = \{a, b, c\}$, and $B = \{\alpha, \beta\}$

then $A \times B = \{(a, \alpha), (a, \beta), (b, \alpha), (b, \beta), (c, \alpha), (c, \beta)\}$

When a set is combined with itself ($A \times A$) it is written A^2. If this notation is extended,

$A^0 = \{\varepsilon\}$

$A^1 = A$

$A^2 = A \times A$

$A^3 = A \times A \times A$

...etc.

The Kleene closure of a set A is denoted by A^* and is the infinite set of strings made from the elements in A, and including the empty string. It is,

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup A^4 \cup \text{ ...etc.}$$

or more succinctly,

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

Another useful definition is A^+ which does not include the empty string,

$$A^+ = \bigcup_{i=1}^{\infty} A^i \qquad A^* = A^+ \cup \{\varepsilon\}$$

5.2.2. Functions

Mathematical functions are defined by two components, a signature and a mapping. The signature specifies constraints and types for the arguments passed, along with a constraint and type for the return value. The mapping specifies the return value for a particular combination of parameters. An example of an abstract function is provided below:

given a function $\delta(s,x) = y$

with a signature $\delta: S \times \Sigma \rightarrow S$

then $s \in S$, $x \in \Sigma$ and the return value $y \in S$.

If a set of values is passed as a parameter instead of an element, then the result is a set, so $\delta(A, \Sigma) = B$. That is, B is the set of values returned by the function δ when every combination of members from A and Σ are passed as parameters.

5.2.3. Quantifiers and other Symbols

The universal and existential quantifiers ($\forall$ and $\exists$) will be used as shorthand as follows:

$\forall$ = "For all .."

$\exists$ = "There exists an ..."

also,

$x \mid y \in S$ = "x such that y is a member of S"

5.2.4. Sequential Machines

The information in this section has been derived from [8].

A sequential machine (SM) is a control structure whose current state and output depends upon the sequential order of its inputs. The next state which the SM will possess is dependent on the current state and any input to the machine. The inputs will be known as **stimuli** during the remainder of this thesis.

The SM may also produce outputs, however, no benefit is gained from their use as a modelling technique for object-oriented programs. Stimuli are used to model calls to particular operations with specifics parameter values, as will be described later. Hence, the stimuli have no direct knowledge of the values involved. Therefore, SM's cannot be used to model output values that are based upon the input values. Furthermore, output values[11] are more easily validated using traditional functional testing approaches. Thus, only the state transition part of an SM will be used to model classes. Automata, which do not possess outputs, are introduced in section 5.2.5.

The operation of the SM can be described by a *transition table* which lists all the possible inputs and their effects on the current state. An SM can also be described by a schematic representation known as a state diagram. Figure 2 shows the state-diagram for a simple SM (without the outputs) for counting from one to four. The arcs represent state transitions; the labels on the arcs (n and r) represent the input symbols that must be applied to the SM to cause the transition.

---

11 Output values are also referred to as the return values of an operation.

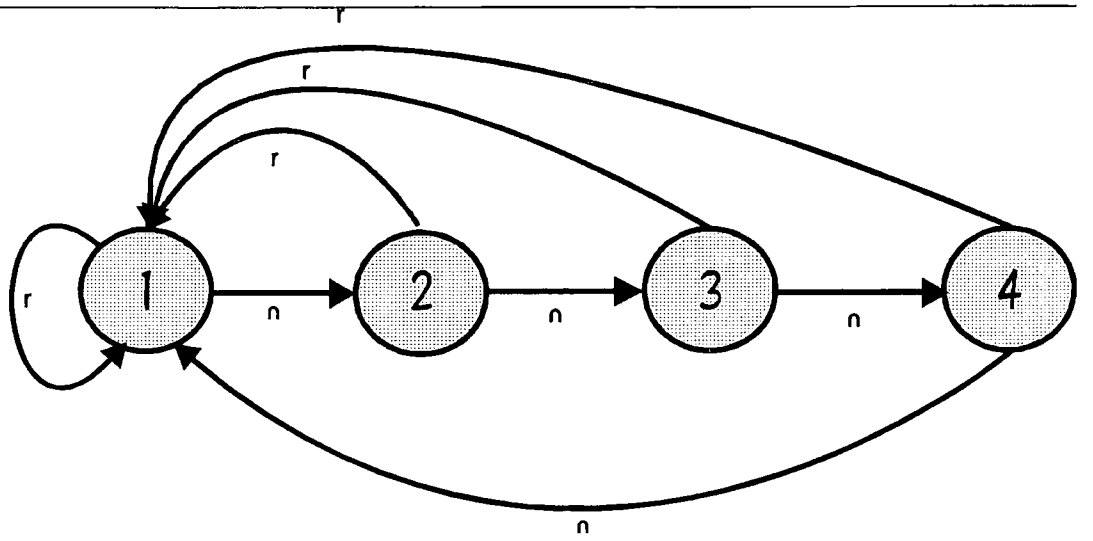

Figure 2. The State-Diagram for an Example Sequential Machine

5.2.5. Automata

An SM consists of two main structures: the transition structure and the output structure. An automata is an SM without any consideration for the output structure. An automata A is defined as the following triple:

$A = (S, \Sigma, \delta)$, where

S is the set of all the states that the automata can possess

Σ is the input alphabet (a set of all the stimuli which can be input to the automata)

δ is the transition function

An automata is known as a Finite State Automata (FSA) if the cardinality of S is finite.

The transition function takes two parameters, the first is the current state s, and the second is the stimuli x; the returned value is the next state y. A transition function is defined by the signature $\delta: S \times \Sigma \rightarrow S$ where $\delta(s,x) = y$.

Now, $\delta(s, xy)$ is defined as $\delta(\delta(s,x), y)$ $\forall s \in S$ and $\forall x,y \in \Sigma$, therefore the function definition has been extended to $\delta: S \times \Sigma^2 \rightarrow S$. Using a similar process the function can be extended to $\delta: S \times \Sigma^* \rightarrow S$, meaning an input string of any length can be applied to an automata by the repeated application of the transition function. This has a special consequence for the testing of classes which will be discussed later in section 5.3.

For the automata in figure 2, the triple is

$S = \{1, 2, 3, 4\}$

$\Sigma = \{n, r\}$

and the transition function (as a table) is

		Input ($x \in \Sigma$)	
		n	r
Current State ($s \in S$)	1	2	1
	2	3	1
	3	4	1
	4	1	1

Table 1. The Transition Table for Figure 2

At present, δ is defined over the whole range of $\delta: S \times \Sigma^* \rightarrow S$, even though $\#(\Sigma^*) = \infty$. This is unusable as the use of Σ^* implies that the class is exhaustively tested with all input sequences. Hence, there is a requirement for a suitable reduction. Initially, the transition function is broken down into separate functions, one for each stimuli. This is defined by $\delta_i(x)$ where,

$\delta_i(x) = \delta(x, i) = y$

$\delta_i : \Im_i \rightarrow \Omega_i$ with $\forall i \in \Sigma^*$

$\Im_i$ will be used throughout the remainder of this document to represent the set of states that an operation can accept as input and Ω_i will be used to represent the set of all states that can be generated by the transition function.

Using δ_i, each input sequence can now be considered as a separate input function. A *complete* input function is one which can accept any valid state as input and an *incomplete input* function is one which cannot accept the whole range of valid states as input. That is,

For a complete function $\delta_i : S \rightarrow \Omega_i$, where $\Omega_i \subseteq S$

For an incomplete function $\delta_i : \Im_i \rightarrow \Omega_i$, where $\Im_i \subset S$ and $\Omega_i \subseteq S$.

If we define $\Im_i + \Psi_i = S$ for an incomplete function, then δ_i has an undefined result for any state that is a member of Ψ_i; therefore Ψ_i is the set of states for which δ_i is undefined.

5.2.5.1. Equivalence Partitions

There are only a finite number of input functions even though $\#(\Sigma^*) = \infty$, due to the equivalence of input sequences. If x and y are any two members of Σ^*, then equivalence ('≡') is defined as

$x \equiv y$ if and only if $\forall s \in S, \delta_x(s) = \delta_y(s)$

A new notation is now introduced to represent the equivalence partitions. [x] is used to denote the equivalence partition with x as a member. However, any member of the partition can be used to represent the class as they are all equivalent. As all the input sequences in the equivalence partition [x] will have the same input function as x (from the definition of equivalence), no change in notation is required as $\delta_x = \delta_{[x]}$.

Software testing uses a similar technique known as equivalence partitioning for reducing the range of inputs that must be passed to functions (see [106] for more details). However, equivalence partitioning views the object under test from a specificational point of view not taking into account extra information that is available from the design. The application of this technique and other functional testing techniques to object-oriented programs will be discussed in the chapter 6.

5.2.6. Summary of Symbols Used

This section summarises the notation that has been described so far.

5.2.6.1. Set Theory, Quantifiers and Other Symbols

Set theory symbols:

$\varnothing$ represents the empty set

ε represents the empty string of symbols

#(S) is the cardinality (number of elements) of the set S

$A \times B$ is the Cartesian product of A and B

A^n is expanded to $A \times A \times A \times \ldots$ n times

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

$$A^+ = \bigcup_{i=1}^{\infty} A^i$$

$$A^* = A^+ \cup \{\varepsilon\}$$

Quantifiers:

$\forall$ = "For all .."

$\exists$ = "There exists an ..."

Other symbols:

x | y$\in$S = "x such that y is a member of S"

5.2.6.2. Sequential Machines and Automata

A Sequential Machine (SM) is a control structure with a state. The state undergoes transitions in response to input *stimuli*. The transitions can be describes by either a state diagram, or a transition table, or a transition function (they are synonyms for each other). The nodes of a diagram represent the states. The arcs represent the transitions and are labelled with the stimuli that cause it. The SM may also have output which are also labelled on the arcs.

An automata is an SM without regard for the outputs, it is defined by a tuple:

$A = (S, \Sigma, \delta)$

where S is the set of all states that the object can possess, Σ is the set of all input symbols, and δ is the transition function. δ is defined with the signature $\delta : S \times \Sigma^* \rightarrow S$ and is broken down in a transition function for each stimuli (δ_i) which are defined as:

$\delta_i(x) = \delta(x, i) = y$

$\delta_i: \Im_i \rightarrow \Omega_i \quad \forall i \in \Sigma^*,$

where $\Im_i$ represents the set of valid input states for the operation, and Ω_i is the set of output states that can be generated by the class.

A complete function is one where $\Im_i = S$, and an incomplete function is one where $\Im_i \subset S$.

Although $\#(\Sigma^*) = \infty$, a set of equivalence partitions can be generated. A partition consists of all those input stimuli that cause the same transition and therefore are equivalent. [x] is used to denote the partition which has x as a member. Any member of the partition can be used to represent it as all members are equivalent. The number of partitions is finite.

5.3. The Modelling of Classes As Automata

This section relates the theoretical concepts of automata to the reality of classes using examples for clarification.

```
class count
{
   public:
      count()          // constructor to ensure initial state is valid
      {
         iStore = 1;
      };

      void reset()       // stimuli 'r'
      {
         iStore = 1;
      };

      void next()      // stimuli 'n'
      {
         iStore ++;
         if (iStore > 4)
            reset();
      };

   protected:
      int iStore;
};
```

Figure 3. An example class for counting from 1 to 4

The example in figure 3 shows a possible implementation of the automata in figure 2 (see section 5.2.4.).

The operation `next()` is used to represent the stimuli 'n', and the operation `reset()` is used to represent the stimuli 'r'. The constructor - `count()`, ensures that the initial state of an object of the class would be valid. The starting state for the constructor is undefined, as would be the finishing state of any destructor.

To facilitate the following comparison between the theoretical automata that the class is based on, and the untested class, the class itself will be modelled as an automata. The following definitions will be used:

The theoretical automata $T = ({}_T S, {}_T \Sigma, {}_T \delta)$

The actual automata $A = ({}_A S, {}_A \Sigma, {}_A \delta)$

T is the intended automata derived from the specification and the design of the class. A is the automata that exists in the form of the class's code.

Testing is used to show that the actual transition function matches the theoretical one for ${}_A\delta : {}_T S \times {}_T \Sigma \rightarrow {}_T S$ and hence it can be expanded to ${}_A\delta : {}_T S \times {}_T \Sigma^{+} \rightarrow {}_T S$. This means that using any sequence of operation calls, no undefined state can be generated.

Note that Σ^+ is used instead of Σ^* as it does not include the empty string of stimuli (ε). The technique which will be described relies on the sequential nature of automata. The technique relies upon the state of an object changing only when a stimulus is applied. Therefore, a restriction must be applied disallowing any automata to change state without the application of stimuli. The technique to be outlined cannot be used to model active processes such as those found in parallel languages. Testing classes modelled as automata involves the creation of a specific state, then the execution of the test case involving the application of a stimulus, and finally the validation of the final state achieved. This must be performed in a controlled manner, with the state changing only on the application of the stimuli applied by the tester. Hence the need for the restriction. The application of state-based testing to parallel languages is discussed more in section 10.3.

The extension of ${}_A\delta$ from ${}_A\delta:{}_T S \times {}_T\Sigma \rightarrow {}_T S$ to ${}_A\delta:{}_T S \times {}_T\Sigma^+ \rightarrow {}_T S$ is easily explained. If ${}_A\delta$ cannot generate a state t such that $t \notin {}_T S$ then because of the definition for ${}_A\delta(t, xy)$, ${}_A\delta$ cannot generate an undefined state by the repeated application of stimuli. If the transition function ${}_A\delta$ is validated for every state $s \mid s \in {}_T S$ with a single stimulus $x \mid x \in \Sigma$, then the transition function has effectively been validated for all states with any input sequence $y \mid y \in \Sigma^+$.

A stimuli[12] equates to a call of an operation with a particular combination of parameter values. If the transition function has been validated for every state transition using a single input stimulus then all operations of the class must have been validated with all their respective input states. The states generated by one operation must be checked for a match as input states for other operations (this is discussed in more detail in section 6.). If this is combined with the definition of ${}_A\delta(s, xy)$, then the class has been validated for all sequences stimuli.

It is assumed that there is a mapping from the theoretical states (${}_T S$) to the actual states (${}_A S$) used by the class. The actual states involves a combination of values from each of the attributes of the class. Using the examples in figures 2 and 3, the mapping in table 2 is generated.

---

12 The analysis of stimuli is discussed in section 5.5.

${}_{T}S$	${}_{A}S$
1	iStore = 1
2	iStore = 2
3	iStore = 3
4	iStore = 4

Table 2. A mapping from theoretical states to actual states

It may be possible to represent a member of ${}_{T}S$ by more than one combination of attribute values. This produces at best a one to one mapping and at worst one to many mapping. It is assumed that there is a one to one mapping of the set of stimuli (${}_{T}\Sigma$) to operation calls with specific parameters, that is, ${}_{A}\Sigma \equiv {}_{T}\Sigma$. These will be discussed further in section 5.5.

With ${}_{A}\Sigma \equiv {}_{T}\Sigma$ and ${}_{A}\delta({}_{T}S, {}_{T}\Sigma) = {}_{T}S$, the class conforms to the theoretical automata. Also, it is not possible for a state transition to occur which would place an object of the class into an invalid state. For ${}_{A}\delta({}_{T}S, {}_{T}\Sigma) = {}_{T}S$, we must have used the mapping of ${}_{T}S$ to ${}_{A}S$ when choosing particular combinations of values for the attributes to represent each member of ${}_{T}S$. Hence, we have actually demonstrated that ${}_{A}\delta({}_{A}S, {}_{A}\Sigma) = {}_{A}S$. This does not however guarantee that the class is free from errors.

As mentioned above, the mapping of ${}_{T}S$ to ${}_{A}S$ may be a one to many mapping. This means that there are a number of different combinations of values for the attributes that could be chosen to represent each member of ${}_{T}S$. The demonstration of ${}_{A}\delta({}_{A}S, {}_{A}\Sigma) = {}_{A}S$ would be performed using only one particular combination of values representing each state. Other combinations of values may cause errors. The more stimuli that are used, the fewer errors can exist in the class; the number used depends upon the testing resources available. The analysis of stimuli is addressed in section 5.5. State-based testing should be used in conjunction with other testing techniques. This is discussed in section 6.

Testing ultimately validates an implementation against its specification. Although it may also validate it against its design (as described here). So far, no assumption has been made about the validity of the states used in the transition table. If a class is design to 'fail', then as long as the 'failure' is well defined, then the class can be validated using this technique. The transition table can contain any states as long as they are well defined in terms of the attribute values they represent.

5.3.1. Special Cases - Constructors and Destructors

Constructors and Destructors are two special types of operation. Constructors are called when the object is in an undefined state with the task of initialising it to a known state. Destructors perform the opposite, they take an object in a valid state and perform any tidying up necessary such as returning allocated memory to the memory heap. Destructors leave the object in an undefined state. Both of these type of operations must be catered for.

A special symbol θ will be used to represent the undefined state that is provided as input to a constructor. It is also the output state from the destructor. This state will not be directly testable. It is provided for easing the expression of states when concerned with special operations.

Using the example class in figure 3, the symbol **c** will be used to represent a call to the constructor `count()`. Although no destructor (`~count()`) was defined for the class because there were no actions that need to be performed to tidyup a `count` object, a destructor is automatically provided by the compiler. The letter **d** will be used to represent a call to the destructor. The transition table for the class (table 3) is obtained from table 1.

		Input (x∈Σ)			
		c	n	r	d
Current State (s∈S)	θ	1			
	1		2	1	θ
	2		3	1	θ
	3		4	1	θ
	4		1	1	θ

Table 3. The Transition Table for the Class in Figure 3

As can be seen in the table, a number of entries are simply blank. This means that the table is not defined for a transition from that state with that stimuli. In terms of the class, the constructor of a class cannot be called once an object has been created. If the object has not been initialised, the results of call to the other operations cannot be defined.

The theoretical automata for the class in figure 3 is now:

$_{T}S = \{\theta, 1, 2, 3, 4\}$

$_{T}\Sigma = \{c, n, r, d\}$

and the input and output sets are:

$_T\Im_c = \{\theta\}$ $\quad$ $_T\Omega_r = \{1\}$

$_T\Im_n = \{1, 2, 3, 4\}$ $\quad$ $_T\Omega_n = \{1, 2, 3, 4\}$

$_T\Im_r = \{1, 2, 3, 4\}$ $\quad$ $_T\Omega_r = \{1\}$

$_T\Im_d = \{1, 2, 3, 4\}$ $\quad$ $_T\Omega_d = \{\theta\}$

A theoretical consequence of the addition of the state θ is that all input functions become incomplete functions (they are not defined over the whole set of possible starting states). Nevertheless, this does not affect the application of the technique because in the vast majority of object-oriented programming languages, the constructor cannot be called once the object has been created, and the invocation of any operations prior to the constructor being called is by definition undefined. The incompleteness of the functions has no bearing on the testing, as the testing should validate only the prescribed transitions, it purely a theoretical issue.

As mentioned above, the constructor cannot be called for an object that has already been created

5.4. Derivation of Automata From the Design

So far, this discussion has assumed the availability of a theoretical automata in the form of a specification or design. However, one is rarely available, therefore a method is required for deriving basic state information from a non automata-based design. This will be introduced below.

5.4.1. Substates

The *current state* of an object is the combination of the values from all of its attributes at the current point in time. It is appropriate to define a **substate** to be the value of a particular attribute at a specific point in time. Thus, the current **state** of an object can be redefined to be the combination of all of the object's substates at the current point in time.

5.4.2. Substate-values

With the example in figure 3 there is only a small number of states and values used by the class. A general method is required for reducing the potentially large range of values that each substate could possess down to a more manageable size. Instead of associating every possible value of a attribute with a substate, it is more appropriate to introduce two types of substate-values:

specific substate values - these are values that should be tested for directly within the code and are described in the design (or specification) as being of special significance.

general substate values - these are a group of values that are all considered in the same manner; therefore there is no need to distinguish between them for the purposes of state-based testing.

A substate's values are a collection of specific-substate-values and general-substate-values. This will be illustrated using the example in figure 3 again; the fact that a theoretical automata already exists for the class will be ignored and the substates will be derived from the design of the class. The following substate-values would be chosen for the substate iStore:

1) iStore = 1 (a specific-substate-value for the starting state)
2) 2 <= iStore <= 3 (a general-substate-value covering the intervening values)
3) iStore = 4 (a specific-substate-value)

Figure 4 shows the range of values used by the iStore attribute.

Section **5.7.** provides guidelines for analysing substate values.

If the theoretical automata for this example was used, then four specific-substate-values would have been chosen. However, using the above method causes the combination of two theoretical states into one general-substate-value implying that they are treated in a similar way. This reduces the testing based upon state information that is performed for this example. However, when the ranges of values are greater it provides a more beneficial reduction in the set of states. The technique reduces the potentially infinite set of states down to a more manageable size. This will be discussed further in chapter 9.

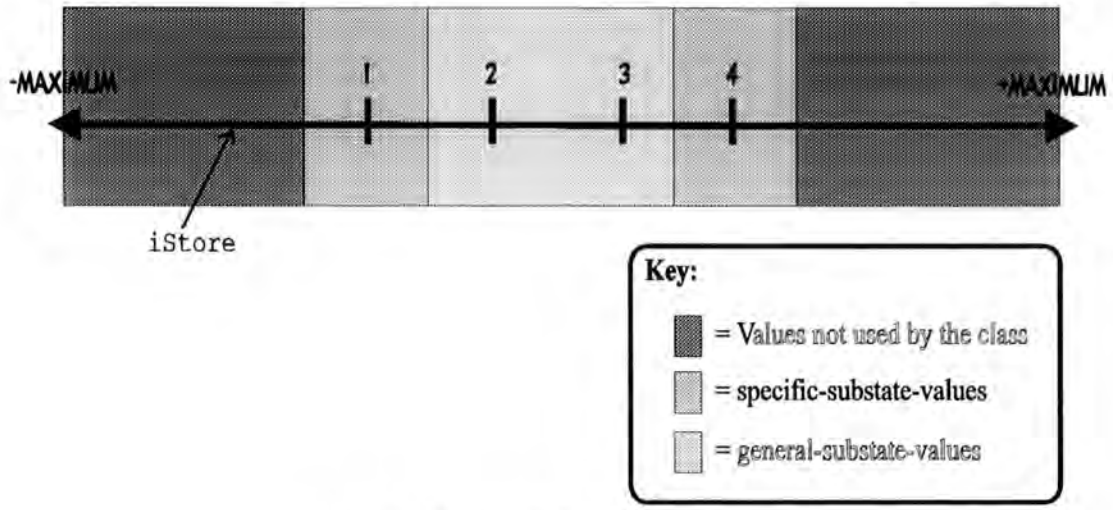

Figure 4. The range of values used by iStore

5.4.3. State Notation

To enable the expression of states and substates in a concise manner the following notation will be used:

S[x] will be used as notation for substate x,

S[x](y) will be used as notation for substate-value y of substate x,

where:

$1 \leq x \leq$ number of substates and

$1 \leq y \leq$ number of substate-values for substate x

Each substate is assigned a unique number starting from 1. Each substate-value is assigned a unique number starting from 1 for each substate.

5.5. The Testing of Classes as Automata

In a similar way that the substate-values were analysed for specific-values and general-values, parameter value ranges can also be reduced. Various functional techniques can be applied to parameters, such as equivalence partitioning and boundary analysis (functional testing techniques are described in chapter 3.). These techniques reduce the potentially infinite range of parameter values to a finite set of partitions of a more manageable size. Partitions are also chosen from the design of the class. A single value is chosen to represent each partition during the testing. The Cartesian product of

the sets of values for each parameter are then used to represent stimuli for the automata. For example, if the parameter values 1 and 3 have been chosen as values for the first parameter, and, 4 and 7 were chosen for the second parameter 2 for use by the operation set_state() then:

Object.set_state(1, 4),

Object.set_state(1, 7),

Object.set_state(3, 4) and

Object.set_state(3, 7) are considered as four separate stimuli.

The class in figure 3 would be tested by invoking each operation with the object in every possible state (this is formalised below). For this example, it would be a relatively trivial task considering the small number of states and the low complexity of the class. A general algorithm is described below which formalises the steps described so far, and can be applied to the vast majority[13] of classes.

First, generate the set of stimuli that will be applied to the object under test:

[1] For every operation **f** of the class repeat step [2]

[2] For every parameter **x** of the operation **f** repeat steps [3] and [4]

[3] Analyse **x** for partitions using the chosen functional testing techniques. $\mathbf{P}_{f(x)}$ is the set of partitions for parameter **x** of operation **f**

[4] Generate the set of stimuli ${}_{A}\Sigma_{f}$ that use operation **f** by generating all combinations of $\mathbf{P}_{f(x)}$ for all parameters

then generate the test cases :

[5] For every operation **f** of the class repeat step [6]

[6] For every stimulus $\sigma \in {}_{A}\Sigma_{f}$ repeat step [7]

[7] For every state $s \in \Im_{f}$ repeat steps [8] through to [11]

[8] Create an object in the state **s**

[9] invoke the operation **f** with the parameter values chosen to represent the stimuli σ

[10] Compare the new state of the object with the state predicted by the transition function (${}_{T}\delta$)

[11] If the states are equal, the test has passed, otherwise the test failed.

---

13 Some classes cannot be tested using state-based testing. If a class has no attributes, then it will have no state.

5.5.1. Exceptions to the Algorithm

There is an exception to step [8]. The tester is unable to create an object in state θ (the undefined state). However, the constructor of the class can still be invoked thus creating and initialising an object. Steps [8] and [9] therefore become one operation.

There is also an exception to step [10]. The tester is unable to validate that an object is indeed in state θ (as it is an undefined state). Separate white box test cases will be required to validate that the destructor has indeed performed its functionality correctly.

5.5.2. Additions to the class

The focus of state-based testing is the validation of an object's state both before and after operation invocation. To facilitate this, a new version of the class under test must be produced allowing the tester access to the attributes in a controlled fashion. There are two sorts of tests against the substates that are required: a "direct test" for a particular substate-value (specific or general), or a "test for a change" in the attribute's value. The use of a "direct test" is self explanatory, however the use of a "test for a change" is more subtle. These are used to detect a change in an attribute value between two values covered by the same general-substate-value. They are also useful for detecting when no change in value has taken place. If an operation should not affect the value of an attribute then it is relatively easy using a "test for a change" to detect when the attribute is affected.

Figure 5 shows an example operation for detecting the current substate-value of S[1], as defined in figure 4.

```
class count
{
   public:
      // .. as before ...

      // a  operation to test the current substate value
      bool TestForSubstate1(state sValue);

      // ... the remainder of the class definition
};

bool count::TestForSubstate1(state sValue)
{
   cout << "TestForSubstate1(" << sValue << ") == ";
   state svCur;
   switch (iStore)
   {
      case 1:
         svCur = 1;
         break;
      case 2:
      case 3:
         svCur = 2;
         break;
      case 4:
         svCur = 3;
         break;
      default:
         svCur = 0;
         break;
   }
   bool bResult = (svCur == sValue);
   cout << bResult << "\n";
   return bResult;
};
```

Figure 5. An example operation for testing the current value of substate 1 (in C++)

The extra operations for implementing a "test for a change" need extra attributes in the class to directly mirror the original attributes. These facilitate the comparison of a change in value.

Figure 6 shows an example implementation of an operation for a "test for a change". The mirror attribute is initialised by calling the operation with `true`[14] as the parameter. The parameter in the example has a default value, so calling it with no parameter will "test for a change" in the appropriate attribute.

14 `true` and `false` are two default instances of a boolean class written by the author.

```
class count
{
   public:
      // .. as before ...

      // a  operation to test the current substate value
      bool TestForSubstate1Change(bool bInit = false);

   protected:
         // as before ...
         int iStore;

         // the mirroring attribute ..
         int iStoreMirror;
};

bool count::TestForSubstate1Change(bool bInit = false);
{
   cout << "TestForSubstate1Change(" <<
   bInit << ") == ";
   if (bInit == true)
      iStoreMirror = iStore;

   bool bResult = (iStore != iStoreMirror);
   cout << bResult << "\n";
   return bResult;
};
```

Figure 6. An example operation for detecting a change in value of a attribute (in C++)

It is recommended that statements are inserted at the beginning and end of each operation to report to the screen (or to a file) the values of the parameters passed and the values returned by the operations. This should also be performed for the substate testing operations as it provides a useful aid in debugging any errors that might occur by showing a trace during the execution of a test case.

It is usual that some operations of a class must perform more than one task or activity to achieve their desired functionality. The greater the number of tasks, the increased likelihood that a greater number of test cases would be required to validate it. Assertions should be inserted into the code between each task, checking the current state of the object, and validating that the sub-task was satisfactorily performed.

5.6. Extensions to the Basic Algorithm

The simple algorithm outlined previously only generates a basic test suite. Enhancements can be made to extend the algorithm. These will be described in section 5.6.1. with the aid of a more complex example (see figure 7 below).

```
// The class used for the nodes in the list ..
struct list_element {
   list_element * pNext;
   TYPE tItem;
};

// The class to be tested ..
class list
{
   public:
      // the definition of the interface goes here ....
      bool first(void);              // go to start of list ..
      bool next(void);               // go to next entry in list ..
      bool get(TYPE & tParam);             // get the current value
stored in list ..
      // ...

   protected:
      // pTop points to the top of the first element of the
      // list (the top)
      list_element * pTop;

      // pCur points to the pointer to the current
      // list_element
      list_element ** pCur;
};
```

Figure 7. The definition of a linked list class written in C++.

The following substates were chosen:

substate 1 (denoted as S[1]) : The value of `pTop`

substate 2 (denoted as S[2]) : The value of `*pCur`

substate 3 (denoted as S[3]) : The value of `pCur`

Below is a summary of the major design points for the model on which the linked list class is based:

pTop points to the first node in the linked list. If the list is empty then pTop equals NULL.

pCur is a pointer to a pointer to the current node in the list. Therefore to point to the first node in the list, pCur equals the address of pTop, as pTop points to the first node in the list. For pCur to point to the second node, pCur equals the address of

*pTop->pNext, and so on. pTop will always exist, therefore pCur will always be a valid pointer, even when *pCur is NULL.*

From this, the following substate-values can be extracted:

For S[1]:

(1) pTop equals NULL (a specific-substate-value)

(2) pTop is not equal to NULL (a general-substate-value)

For S[2]:

(1) *pCur equals NULL (a specific-substate-value)

(2) *pCur is not equal to NULL (a general-substate-value)

For S[3]:

(1) pCur equals the address of pTop (a specific-substate-value)

(2) pCur is not equal to the address of pTop (a general-substate-value)

If the attribute being considered is a **pointer**[15] (an address of an object, rather than an actual object), then it is likely that the pointer will have at least two substate-values, one specific and one general. The specific-substate-value is likely to be the special value **NULL**[16], and the general-substate-value will cover all other possibilities, that is, any other value except NULL. This is true in the majority of cases, however it is not true all of the time. This is shown by the use of pCur above. pCur does not use the value NULL as the address of pTop is used as its default value instead.

This is discussed more in Section **5.7.**

5.6.1. Data Scenarios

State-based testing is useful for more than merely detecting the change in the state of an object, it can be used to detect the correct construction of a more complex dynamic data structure; for example, a linked list. For such dynamic data structures, it is essential to determine what changes can occur to the structure, and when they can occur. This analysis generates a list of situations which are significant to the model upon which the data structure is based; these situations will be referred to as **data-scenarios** or just **scenarios**.

---

15 Pointers are sometimes known as *references.*

16 NULL is the value assigned to a pointer to show that it has no attached object (in C++/C).

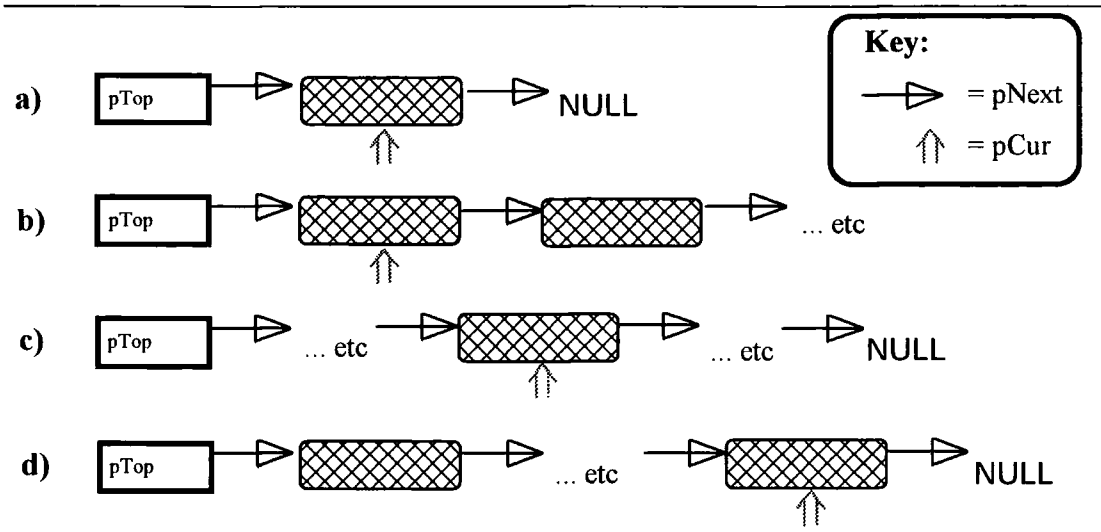

Figure 8. The data scenarios for the operations that act upon the nodes of the list

Using the linked list example class in figure 7, two sets of scenarios are needed: scenarios for the operations which interact with the nodes of the list directly (see figure 8), and scenarios for the operations which act upon the links between the nodes (see figure 9).

The data-scenarios that affect the nodes are : (see figure 8)

a) the only node of the list; that is, no nodes before or after the current node,

b) the first node in a multiple node list; that is, there are nodes after the current one, but none before,

c) a node in the middle of a multiple node list; that is, there are nodes both before and after the current one,

d) the last element of a multiple node list; that is, there are node before the current one, but none after.

The data-scenarios that affect the links between nodes are: (see figure 9)

a) an empty list,

b) before the only element,

c) after the only element,

d) in between two nodes in the middle of a multiple node list,

e) at the end of a multiple node list.

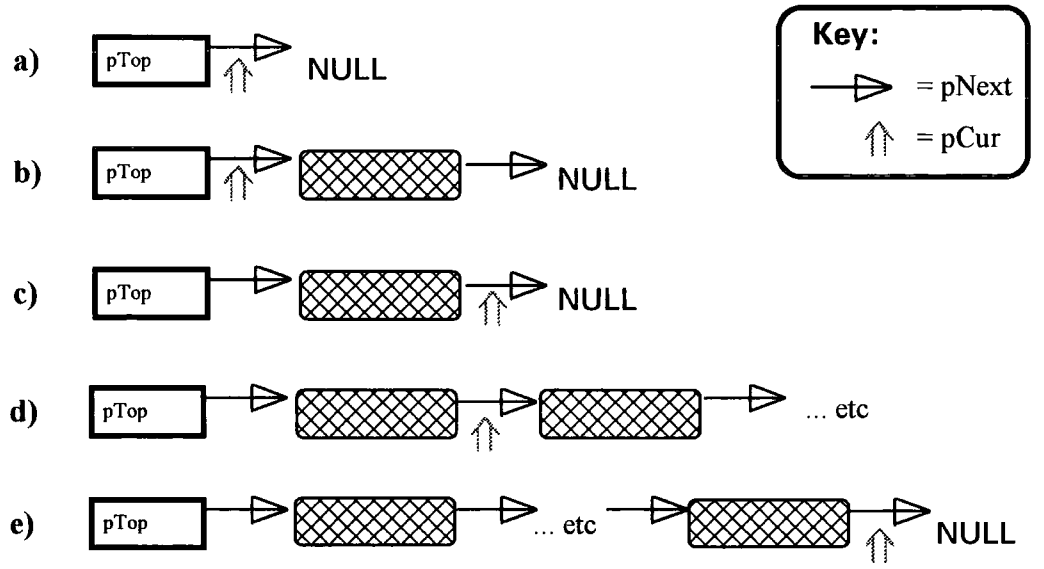

Figure 9. The data scenarios for the operations that act upon the links between the nodes.

Some scenarios from both sets refer to the same physical situation, however, they are described from a different point of view. The two sets are combined to form one set of physical data-scenarios:

i) an empty list,

ii) a single node list with the current pointer at (before) the only node,

iii) a single node list with the current pointer after the only node,

iv) a multi-node list with the current pointer at (before) the first node,

v) a multi-node list with the current pointer at (before) a node in the middle of the list,

vi) a multi-node list with the current pointer at (before) the last node,

These data-scenarios are used as additional test cases to the appropriate groups of operations. However, unless a change in the composition of an object's dynamic data structure is detectable, the results of using the above scenarios will be difficult to verify. This can be achieved by the allocation of additional substates. These will be used to detect the state of various parts of the dynamic data structure. For the above scenarios, the following additional substates are needed:

S[4](*pCur)->pNext

(1) invalid (if *pCur equals NULL it cannot be dereferenced)

(2) (*pCur)->pNext does not equals NULL

(3) (*pCur)->pNext equals NULL

S[5](*pCur)->tItem

(1) (*pCur)->tItem is invalid (if *pCur equals NULL it cannot be dereferenced)

(2) (*pCur)->tItem is valid

The substate values S[4](2) and S[4](3) are relatively self-explanatory; however, S[4](1) has to be included, because `*pCur` has the potential to equal `NULL` rendering the expression `(*pCur)->pNext` invalid[17]. The inclusion of an *'invalid'* substate-value is not only necessary when dereferencing pointers. They are also necessary for all expressions that can potentially generate an error when they are evaluated. For example, if *i* is an index into an array, then the value stored in *i* could potentially evaluate to an 'out of range' error, hence an *'invalid'* value must be included as a special substate-value.

This also applies to S[5], (see S[5](1) above) which seems to indirectly reflect the value of `*pCur` (S[2]) alone. This is because the list is used to store any range of values. It is pointless to detect every single value stored. The ability to 'test for a change' in the value stored is therefore needed. This facility is provided by some of the extra operations that have been added to the class under test (see section 5.5.2.); furthermore, allocating substates and substate-values to the expressions enables them to be tested in a similar manner which is consistent with the other substates. It also facilitates the correct expression of test cases to the tool MKTC. MKTC is a prototype tool that has been developed to aid the evaluation of the state-based testing technique. See chapter 7. for more details.

It is conceivable that even when using the additional substates and substate-values the tester cannot differentiate between the chosen data-scenarios. This is of little consequence as the initial state of the object prior to the execution of the test case is created by the tester, who is able to create an object in any require state. Only changes in the construction need be detected. It should be possible to detect these with the additional substates. It is to circumvent this lack of differentiability between data-scenarios that the tool MKTC has a facility allowing the starting state for test cases to be expressed as a data-scenario, instead of a state description.

Each data-scenario in the list of physical data-scenarios (above) must be analysed for its corresponding state. That is, if an object was created in that particular scenario, what state would it be in ? The following list is produced:

scenario i) an empty list

S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2)

scenario ii) a single node list with the current pointer at (before) the only node,

S[1](2) & S[2](2) & S[3](1) & S[4](1) & S[5](1)

scenario iii) a single node list with the current pointer after the only node,

S[1](2) & S[2](1) & S[3](2) & S[4](3) & S[5](2)

---

17 A NULL pointer cannot be dereferenced as it does not point to a valid object.

scenario iv) a multi-node list with the current pointer at (before) the first node,

S[1](2) & S[2](2) & S[3](1) & S[4](2) & S[5](1)

scenario v) a multi-node list with the current pointer at (before) a node in the middle of the list,

S[1](2) & S[2](2) & S[3](2) & S[4](2) & S[5](1)

scenario vi) a multi-node list with the current pointer at (before) the last node,

S[1](2) & S[2](2) & S[3](2) & S[4](1) & S[5](1)

These should be incorporated into a data-scenario transition table as described in table 4 below.

5.6.1.1. Data-Scenario Transition Table

A slightly different transition table is required to predict the effect of applying the stimuli ($_A\Sigma$) to the scenarios. Instead of being a mapping from a current state to a new state, it is a mapping from a scenario to new state. This is because a scenario is a more specific expression of the construction of a dynamic structure than a state is. For example, using physical scenarios one and two (from above) we can define a partial data-scenario transition table (see table 4).

			Input Symbol		
	State	Initial State	f	n	g
Scenario 1	S[4]	1	1	1	!$
	S[5]	1	1	1	!$
Scenario 2	S[4]	2	2	3	!$
	S[5]	2	2	1	!$

Table 4. An Example Data-Scenario Transition Table (partial)

In table 4 the symbol '!$' is used to represent a "test for no change" in value. A complete table should be defined, however for clarity table 4 only uses two scenarios and two substates. The complete table would contain an entry for every scenario, substate and input stimuli.

5.6.2. Generation of Test Cases

The algorithm presented in section 5.5. formalised the basic process of generating a suite of state-based test cases. The algorithm presented below is an extended version incorporating a formalisation of the steps that are involved in creating test cases based on scenarios.

First, generate the set of stimuli that will be applied to the object under test:

[1] Allocate one substate per attribute.

[2] Determine the data scenarios from the design of the class.

[3] Allocate any extra substates required for the data scenarios to function properly.

[4] Determine the specific and the general-values for all these substates. Include any *invalid* values that are required, as mentioned in section (5.6.1.)

[5] Add operations to the class which will perform a "direct test" for substate-values.

[6] Determine the substates for which a "test for a change" are required. Add operations to the class for their detection. Also add any new attributes required for this.

[7] For every operation **f** of the class repeat step [8]

[8] For every parameter **x** of the operation **f** repeat steps [9] and [10]

[9] Analyse **x** for partitions using the chosen functional testing techniques. Also incorporate partitions derived from the design of the class. $\mathbf{P}_{f(x)}$ is the set of equivalence partitions for parameter **x** of operation **f**

[10] Generate the set of stimuli ${}_{A}\Sigma_{f}$ that use operation **f** by generating all combinations of $\mathbf{P}_{f(x)}$ for all parameters

then generate the test cases :

[11] Analyse the design for the class call graph for inter-operation calls.

[12] Order the operations. Operations which do not call other operations within the class should be tested first. Followed by the other operations of the class, these are determined by working up the call-graph tree.

[13] Use the ordering of operations from step [12]. For every operation **f** of the class repeat step [14]

[14] For every stimulus $\sigma \in {}_{A}\Sigma_{f}$ repeat step [15]

[15] For every state $s \in \Im_{f}$ repeat steps [16] through to [19]

[16] Create an object in the state **s**

[17] invoke the operation **f** with the parameter values chosen to represent the stimuli σ

[18] Compare the new state of the object with the state predicted by the transition function $({}_{T}\delta)$

[19] If the states are equal, the test has passed, otherwise the test failed

[20] For every data-scenario **ds** appropriate for operation **f** repeat steps [21] through to [24]

[21] Create an object in the scenario **ds**

[22] invoke the operation **f** with the parameter values chosen to represent the stimuli σ

[23] Compare the new state of the object with the state predicted by the data-scenario transition-table.

[24] If the states are equal, the test has passed, otherwise the test failed

5.7. Guidelines for Choosing Substate-values

This section is an aid to any tester who will be applying the technique to programs in the real-world. From the previous sections, it will become apparent that the effectiveness of state-based testing is heavily dependent upon the substate-values, both general and specific, that are chosen. In addition the careful choice of scenarios can also enhance the effectiveness of testing. The following list has no particular order, it is a series of observations that have been made over the course of the research. They are provided to aid the tester in applying state-based testing effectively.

As mentioned in section 5.4.2. substate values are split into two categories, general-substate-values and specific-substate-values. These will be found by analysing the low level design of the component under test. Generally, specific-substate-values are located more easily that general-substate-values. Specific ones are usually values that represent specific situations within the state of the object. Either Boolean flags for specific events, or values which have a special meaning in comparison to the other values of the same attribute. The simplest example is an attribute that holds the memory address (a pointer, or reference) of another object. More often than not, the memory values 0 is reserved to signify that the attribute points to no valid object.

General-substate-values can be found in a similar way to specific-substate-values if the design specifically mentions the use of ranges of values that should behave in a similar manner. Otherwise general-substate-values are used to represent the ranges of valid values that are not directly addressed in the design and therefore are expected to behave in a uniform manner (this depends heavily on the completeness of the design). Using the pointer example mentioned above, a specific-substate-value represents the special value 0, therefore there is a general-substate-value that represents all other valid pointer values. It is recommended that general-substate-values be used to represent continuous intervals of values rather than discontinuous. With a continuous range of values, they are easier to detect, and easier to manage during the planning of the test cases.

Any individual value that is mentioned directly in the design must be used as a specific-substate-value. It is highly likely that these values will be tested for and used directly within the code and therefore they are a good candidate for using in test cases. General values are more difficult to accommodate because not all the values in the range can be used during testing due to the exhaustive and infeasible nature of using all values. Values must be chosen to represent the ranges, one technique would be to apply boundary analysis (a black box testing technique) to the ranges which would yield a value in the middle of the range, and one value at each end of the range. The practicality of applying testing in this manner requires that the same test case been executed with each of the three values and the resulting outputs compared for equality.

Scenarios are more difficult to analyse because they vary with the type of object being tested. If the object under test uses dynamic data structures then there will be scenarios that cover all of the major formations the dynamic data structure can be in. For example a tree structure would have a minimum of 4 scenarios: an empty tree, a tree with only a left branch, a tree with only a right branch, a small tree with both branches, etc. When using pointers, a substate value is normally reserved to take account of a 0 (or NULL) pointer which therefore cannot be dereferenced.

For non dynamic data structures, scenarios are usually used to represent conceptual states that the object can have. These include valid and error states (an error state is where the object has detected an error and is therefore not in a normal mode of operation). Scenarios can also be used to represent situations and events that involve external objects. An example would be if the object under test read characters from a file, then some scenarios would be: an empty file, no file, a file with data, a file with the file pointer at the end of the file, etc. As yet there is no definitive way of determining scenarios, this is an area that must be addressed within the scope of future research.

5.8. Summary

This chapter has presented the theory and practice for the generation of a state-based suite of test cases. State-based testing validates the interactions that occur between the operations and the state of an object.

Automata were used to model the internal representation of a class with an emphasis on the state transitions that could occur. A mapping from the physical (classes) to the theoretical (automata) was also presented. A state transition table was used to model the effects on the state, of invoking

operations. Constructors and destructors were also discussed because their use involves an undefined state for the object either before the operations invocation (constructors) or after (destructors).

Not all classes will be modelled as an automata during the design phase, hence automata and state-transition information may not be readily available. A technique was presented for the derivation of an automaton from a non-automaton based design.

A simple algorithm for the creation of a basic suite of state-based test cases was presented. It incorporated analysis of parameter values to represent the stimuli. This algorithm is adequate for simple classes which do not use dynamic data structures.

The method was then extended to include extra information which can detect a change in the construction of a dynamic data-structure. Data-scenarios were introduced for handling situations which are important to the correct construction and manipulation of data-structures. These were then incorporated into the algorithm producing a more complete description of the generation of a suite of state-based test cases.

Chapter 6.

The Testing Process

6.1. Introduction

The previous chapter described both the theory and practice of state-based testing. This chapter will discuss its integration into the standard practices of unit and integration testing.

First, a minor adaptation is described that allows traditional black and white-box testing techniques to be used with object-oriented programs. This is outlined in section 6.2. It is followed by a discussion of unit testing (section 6.3.). A combined technique of black-box, white-box and state-based testing is proposed, and discussed in terms of the different emphasis provided by each of the techniques.

The potential exists for a reduction in the number of test cases when considering classes that are defined using inheritance. The original algorithm proposed by Harrold et al. [55], has been extended to include test suites for state-based testing. This is discussed in section 6.4. In addition, state-based testing can be used to detect the change in context that can occur when an operation is inherited. This is also discussed in this section.

Integration testing of objects and the state-transitions in parameter objects is discussed in section 6.5.

Section 6.6. summarised the information provided in this chapter.

6.2. Adaptation of Traditional Testing Techniques to Object-Orientation

Both black and white box techniques rely upon the process model in figure 10. Black-box techniques validate the external view of the unit, whereas white-box techniques validate the internal (implementational) view of the unit.

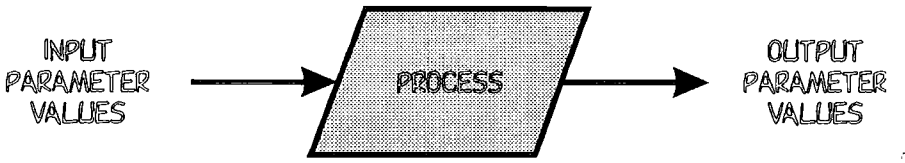

Figure 10. The Von-Neumann model of processing

For it to be possible to use these techniques with the object-oriented programming style, a minor adaptation is required. The current state of the object should be considered as an input, and the new state of the object as an output. This is shown in figure 11.

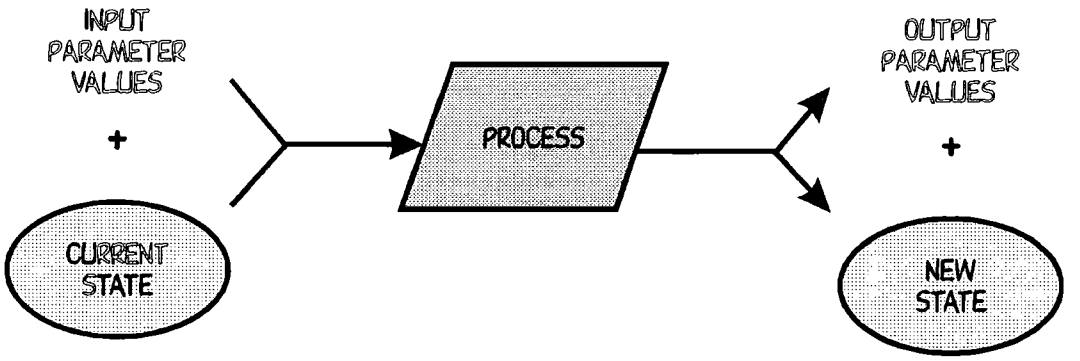

Figure 11. Adapted Von-Neumann process model

6.3. Unit Testing

The IEEE define unit testing to be the:

> "testing of individual hardware or software units or groups of related units."[71]

It is the testing of each separate unit of a program in isolation. With more traditional procedural languages, the size of a unit is usually the function, or procedure; whereas, with object-oriented programs it can be either: the operation, class, or cluster.

It is difficult and time-consuming to isolate an operation from its surrounding class and then to write test drivers and stubs (which themselves require testing) to replace the code removed. The functionality of the stubs and drivers already exists and the majority of the code required is already provided in the form of other operations of the same class. Hence, the class is the smallest sensible unit of test [138, 10, 13] (see chapter 4 for a detailed discussion).

Units are tested in isolation by surrounding them with test stubs and drivers. Test stubs are replacements for the classes that *receive* messages sent from the unit under test. The receiving classes are replaced by simplified simulations of the class allowing the tester to monitoring any values that the class under test may pass to other classes. Test drivers are replacements for the classes that *invoke (or send)* messages to the class under test. Test drivers must create an object of the class under test, invoke its operations with the appropriate parameters thus creating the initial scenario for the test case, perform the test case, and validate any results.

This technique has a number of draw-backs:

1) if a parameter object belongs to the same class as an attribute, a number of related problems occur. Two versions of the same class have to be in existence at the same time. They must have compatible interfaces, and they have to be compatible within the type system of the language used for their implementation.
2) The use of test stubs and drivers involves the writing of additional code. This additional code also requires testing to ensure that it performs correctly.
3) Because of the increase in size and complexity of a class when compared to a single function, it is more difficult to write test drivers and stubs to simulate the required functionality.

A greater advantage will be gained if unit testing involves some of the tasks more traditionally associated with integration testing. The structure of a program can be described by an instantiation tree showing which classes are used as attributes and parameters of each class. Figure 12 shows and example of an instantiation tree[18]. The tree should be used incrementally to test each unit, starting with the bottom layer, that is, StringOfTokens, StateValues, SubstateRef, and continuing up the tree to lex, substates, etc.

---

18 This example was taken from the tool MKTC which will be described in the next chapter.

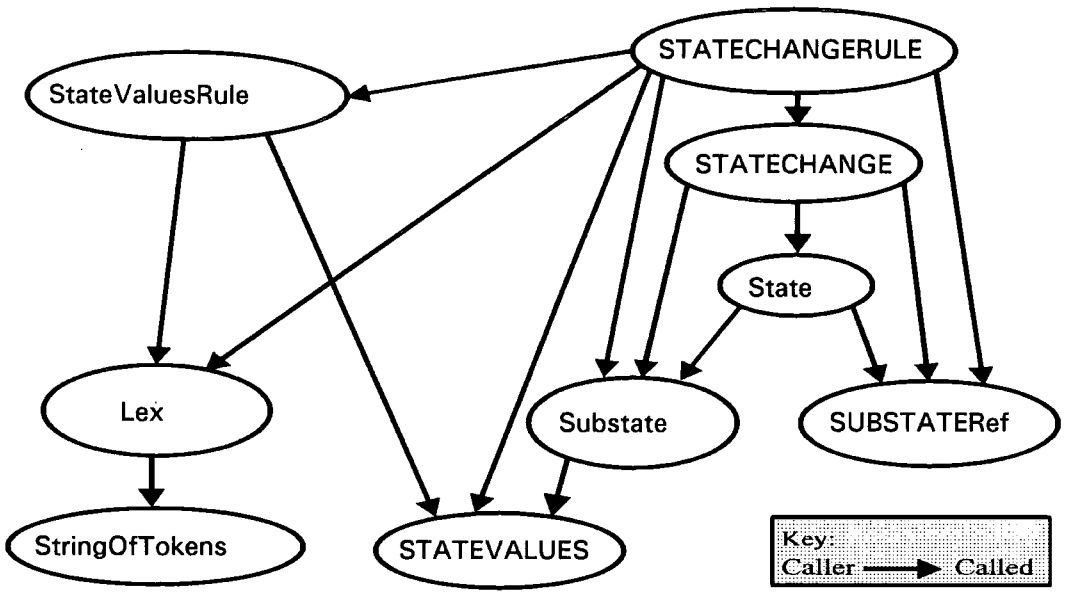

Figure 12. An example of an instantiation tree (from the tool MKTC)

If testing is performed in this order, there is less of a requirement for test stubs, as the classes for parameter objects and attributes will have already been unit tested.

An exception exists; when dealing with external stimuli, it is likely to be more cost effective to write the test stubs for unit testing, therefore facilitating a simulation of the interaction with the outside world.

Another drawback of this method is that it requires the system to be built from the 'bottom up'. Higher level classes will not be tested until much later on in the life-cycle, hence increasing the cost of any major modifications that must be made due to the evolution of the design. Also, testing of the higher level classes must wait for the testing of the classes lower down the hierarchy to be completed. Thus unit testing could not be performed directly in synchronisation with development.

6.3.1. A Combination of Techniques

The focus of unit testing is both external and internal. The external view of the class must be validated to ensure that it performs according to its specification. The internal view of the class must be validated to ensure that it performs according to its design and ultimately its specification. Although each class is viewed as a black-box which provides an interface through which all interaction must take place, truly exhaustive black-box testing is infeasible. Hence, the implementation of the class must also be validated using other techniques such as state-based testing and white-box testing techniques.

During traditional unit testing, it is recommended to use a combination of white-box and black-box techniques [154, 147, 60]. This is because of the difficulty in producing a single technique that is applicable to all programs, is easy to use, and 100% effective in finding all errors. The techniques are used in conjunction with a coverage measure (these were outlined in chapter 3.). First, black-box testing is used to validate the external view of a unit against its specification. Alone, this is unlikely to attain the required coverage level. Hence, white-box testing is then used to 'soak-up' any remaining coverage, until the required level has been achieved. This combination of techniques is commonly referred to as grey-box testing.

In traditional procedural languages, the path executed through a procedure is determined by the value of any parameters and global variables at the time of the procedure call. The values are used to affect the control-flow within the function and so produces the required result. In object-oriented programs, the state of the object persists between calls of its various operations, providing an additional influence affecting the control-flow of the execution.

State-based testing is not a 'stand-alone' technique as it is very unlikely to even produce 100% statement coverage. Hence it must be used in conjunction with other techniques. It is proposed that a combination of black and white-box techniques be used to ensure that the remaining functionality not tested by the state-based test suite, is validated.

6.3.2. Testing Emphasis

As mentioned above, a combination of three types of testing techniques will be used to validate object-oriented programs: black-box techniques, white-box techniques and state-based testing. Each of these has a slightly different emphasis. Black-box techniques use a purely external view of an object, whereas white-box techniques use an implementational (internal) view of an object. In contrast, State-based testing validates the interactions between the operations and the state of an object.

Black-box testing views the current state of an object conceptually. The states used do not directly involve the values of the attributes, they involve situations that are considered conceptually important to the object under test. These are similar to the data-scenarios described in a previous chapter, however, the main emphasis is on the values passed as parameters, and the values returned. This is shown in figure 13.

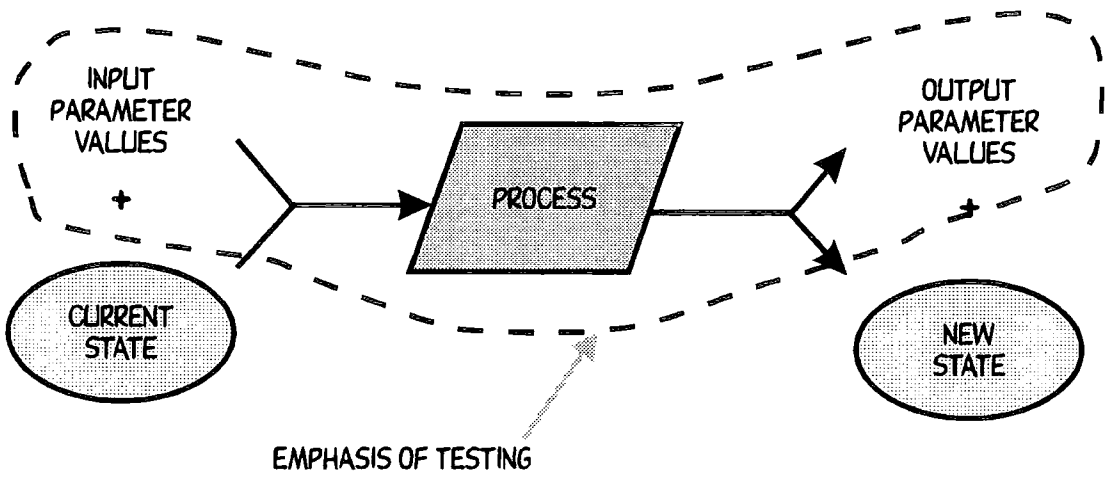

Figure 13. Emphasis of Testing for Black-Box Testing

State-based testing however, is concerned with the actual values stored by the object's attributes. It is not directly concerned with the validation of particular combinations of parameter values, although these values are used to validate the state-transitions of the object. This is shown in figure 14.

An object's attributes are used for two distinct purposes: **control-information**, and **data-storage**. Control-information is used either to 'remember' events, or to communicate them to other operations. Whereas data-storage is used either to store information supplied by the caller, or to store the result of a call to another operation, therefore allowing access to information that would otherwise be transient by nature. It is the use of control-information that is tested with state-based testing.

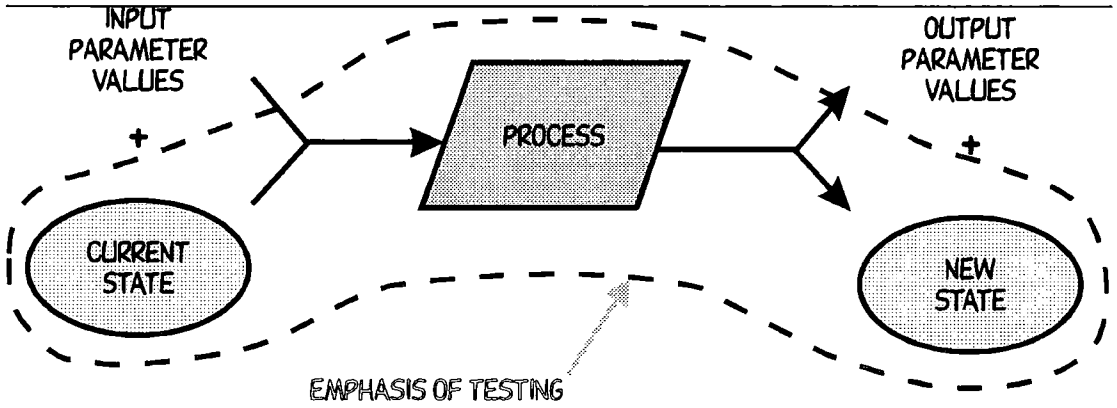

Figure 14. Emphasis of State-Based Testing

An overlap will exist between the suites of test cases that are executed. Some of the conceptual states used for the black-box testing may correspond directly to states that are used by the state-based testing suite. However, the state-based testing suite does not validate any of the values returned by an operation. This must be performed by the black-box test suite. Hence, a certain level of overlap is necessary.

It is recommended that the order of testing is as follows:

1) state-based testing and black-box testing,
2) white-box testing.

No particular benefit is gained when state-based testing is performed before black-box testing, or vice-versa. Although the analysis of states for state-based testing may be useful when considering the conceptual states needed by the black-box testing. The tester should choose the order most appropriate for their situation. White-box testing is placed last in the order so that it may be used to execute those paths not covered by the other two techniques.

6.4. Inheritance

Classes may be defined as an extension or adaptation to a previously defined class, using inheritance. Two main techniques exist for testing child classes (as reviewed in chapter 4.):

1) class flattening. This involves flattening the class hierarchy so that the class is tested as if it provided all the operations. With a large class hierarchy, this can be very expensive in terms of retesting code that has already been validated in a parent class.

2) The application of an incremental testing algorithm. The algorithm reduces the testing of operations inherited from parent classes by analysing their access to and use of attributes.

An algorithm for 2) was proposed by Harrold et al. [55], it uses black-and white-box techniques. However, it does not take into account the state-based testing technique outlined in a previous chapter. The algorithm has been adapted by Turner and Robson [139], and is described below.

6.4.1. Incremental Testing of Classes

The original algorithm uses a test history for storing and controlling the execution of the test cases for each class. As each class is analysed for testing, its test history is formed by combining existing tests cases from its parent's test histories with new ones designed specifically for the new class.

The algorithm uses a classification of operations to decide whether to reuse test cases from a parent class, or to generate new ones. There are six categories of operations:

NEW - The operation is defined in the current class, but not in any of the parent classes. This includes operations that have the same name as an operation from a parent class, but the types of the arguments are different. Thus overloaded operations are treated as completely separate operations.

VIRTUAL_NEW - The operation is not defined in any of the parent classes. The implementation provided in the current class is incomplete.

RECURSIVE - The operation is defined in a parent class and is inherited by the current class with out any redefinition.

VIRTUAL_RECURSIVE - The operation is defined in one of the parent classes, however the implementation is incomplete. The operation is inherited from the parent class without any form of redefinition.

REDEFINED - The operation is defined in a parent class and the current class. A complete implementation is provided in both classes, however the one provided in the current class supersedes the one from the parent class.

VIRTUAL_REDEFINED - The operation is defined in one of the parent classes. However its implementation is incomplete. A complete implementation is provided in the current class.

The above descriptions use terminology similar to that used by Harrold et al. However, a level of ambiguity exists in their description of an implementation as being 'incomplete' (see VIRTUAL_NEW, VIRTUAL_RECURSIVE and VIRTUAL_REDEFINED). The above definitions are more consistent with object-oriented programming if this description is altered. As the names suggest (VIRTUAL_NEW, etc.), some of the operations use dynamic binding to allow the correct implementation to be called at run-time. It is felt that it is this criteria that is used for the above categorisation. Therefore, the new meaning will be assumed throughout the remainder of this discussion.

A graph is constructed for each class, it contains both its operations and its attributes. Each node represents either an operation, or an attribute, each arc represents a message (either a call to another

operation, or an access to an attribute). It is possible for the graph to consist of a number of disconnected sub-graphs. The graph is used to determine the interdependencies between attributes and operations.

The test history consists of two tuples for each operation. The first tuple stores the test cases for unit testing the operation, it consists of: one field for storing the operation name, another the white-box test cases, and the other the black-box test cases. A separate tuple is used to store the integration test cases. A flag is incorporated into each tuple which determines whether the test cases require execution on the current test run. All new test cases are added with this flag set to TRUE.

The tuples must be extended to include the test cases generated by state-based testing.
A number of terms are required for the description of the algorithm:

TS = The black-box test suite for the testing of operations in isolation.

TP = The white-box test suite for the testing of operations in isolation.

TIS = The black-box test suite for the integration testing of operations.

TIP = The white-box test suite for the integration testing of operations.

TSB = The state-based testing suite for the testing of operations in isolation.

TISB = The state-based testing suite for the integration testing of operations.

Each operation of the class is dealt with individually. Assuming the operation is called A, the adapted algorithm is (the alterations are shown in bold italics):
for NEW or VIRTUAL_NEW operations,

1) Generate ***TSB***, TS, and TP for A
2) Add (A, ***(TSB, Yes)***, (TS, Yes), (TP, yes)) to the test history
3) Integrate A into the class graph
4) Generate ***TISB***, TIS, and TIP for A
5) Add (A, ***(TISB, Yes)***, (TIS, Yes), (TIP, Yes)) to the test history

for RECURSIVE or VIRTUAL_RECURSIVE operations,

1) if A interacts with NEW, or VIRTUAL_NEW operations, flag appropriate integration tests for execution.
2) if A interacts with REDEFINED, or VIRTUAL_REDEFINED operations, flag appropriate integration tests for execution.

for REDEFINED or VIRTUAL_REDEFINED operations,

1) Generate ***TSB***, TP for A
2) Reuse TS for A (if it exists), otherwise generate TS (that is, reuse TS tests from parent where appropriate)

3) Add (A, ***(TSB, Yes)***, (TP, Yes), (TS, Yes)) to the test history

4) integrate A into the graph if not already there.

5) generate ***TISB***, TIS and TIP for A

6) Add (A, ***(TISB, Yes)***, (TIS, Yes), (TIP, Yes)) to the test history.

TSB (the state-based test cases for the testing of an operation in isolation) are generated by the technique outlined in the previous chapter.

TISB (the state-based test cases for the testing of the interaction between operations) are a simple validation that the states produced by one operation are the states required by another operation to produce the required functionality. These test cases are usually non-executable and hence are usually performed manually.

6.4.2. Variations in the Use of the State

For each new attribute defined by the class, the specification and design must be analysed for the *substate-values*, as described in the previous chapter. This process must also be carefully repeated for inherited attributes in the context of new operations. Child classes which have access to the attributes of a parent class may have altered/increased the number of substate-values for a particular substate, therefore requiring the retesting of operations from the parent class. This is to satisfy the anti-decomposition principle [147, 112] which deals with a change in the context of operations.

No alteration occurs to the context of inherited operations if any new substate-values for an inherited substate were originally values covered by a general-substate-value. This ensures that any new states that will be generated by the operations of the derived class are still compatible with the original operations. However, if any of the new substate-values were values not previously covered by any general-substate-values then all the operations from the parent class that interact with the affected substates must be partially retested.

For example, a parent class:

```
Class Parent
{
   public:
      // foo uses the data member iValue
      void foo(void);

      // bar uses the data-member iValue and the data-member sName
      void bar(void);

      // boo uses the data-member sName
      void boo(void);

   protected:
      // S[1](1) specific-value = 0
      // S[1](2) specific-value = -1
      // S[1](3) general-value = 1 -> 50
      int iValue;

      // S[2](1) specifc-value = ""
      // S[2](2) general-value = any combination of characters
      string sName
};
```

Figure 15. An Example Parent Class

and a child class can be defined as:

```
Class Child : public Parent
{
   public:
      // foo_be_doo uses iValue. It adds the specific-value 51
      void foo_be_doo(void);

      // loop_de_loop uses sName. It adds the specific-value "Chris
Turner"
      void loop_de_loop(void);
};
```

Figure 16. An Example Child Class

As can be seen above, `Child` inherits all the operations and attributes from `Parent`. It also defines two new operations, `foo_be_doo()` and `loop_de_loop()`. `loop_de_loop()` will be considered first.

`loop_de_loop()` introduces the new specific-substate-value, `sName = "Chris Turner"`. This was previously part of a general-substate-value, hence none of the inherited operations require retesting.

Now, lets consider the operation `foo_be_doo()`. In contrast, the new specific-substate-value, `iValue = 51`, was not previously part of a general-substate-value. The range of values that

`iValue` uses has been extended. However, the operations `foo()` and `bar()` have only been validated for the previous range of values. Hence, both operations require retesting with the new set of states that can be generated with `iValue = 51`. All operations requiring retesting must be tested with all possible input stimuli. This process must be repeated for all substates whose range of values have been extended.

Using the algorithm presented above, it is possible to unit test child classes without unnecessarily retesting inherited operations.

6.5. Integration Testing

The IEEE define integration testing as

> *Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them. [71]*

Integration testing involves incrementally combining components and validating their interactions. Their interactions are validated by carefully monitoring the values and objects passed between objects. Unit testing concentrated on the interactions and algorithms within an object, whereas integration testing focuses on the interaction between objects.

Integration testing involves the careful creation of specific states for all objects involved and the careful monitoring of messages that are sent between them. So far, automata (or classes) have only been considered in isolation, now the interaction of automata must be considered. To integration test a number of objects, the emphasis must be on the state transitions that are observable in all the parameter objects passed and returned from an operation. This is in contrast to unit testing where the state of parameter objects are used to affect the state of the object under test.

Integration testing of three classes, where A and B are parameters to class C, would need test cases that are similar to:

1) Create an object - **a** of class A in its required starting state
2) Create an object - **b** of class B in its required starting state
3) Create an object - **c** of class C in its requires starting state
4) perform the actions required for the test case
5) validate the final state of **a**
6) validate the final state of **b**
7) validate the final state of **c**

Although the validation of **c** is not strictly necessary as it will have already been performed during unit testing, the extra validation is of little cost and will help to detect errors that are intermittent (i.e. not consistently reproducible).

6.6. Summary

This chapter has discussed the incorporation of state-based testing into the unit and integration testing phases of software validation. Traditional testing techniques have been adapted by the inclusion of specific information about the current state of the objects under test. This facilitates their use for testing object programs.

A simple extension to the incremental inheritance testing algorithm of Harrold et al. was presented. The extension involves the addition of state-based test suites in co-operation with an enhanced ability for detecting changes in the context of inherited operations.

Chapter 7.

A Suite of Prototype Tools

7.1. Introduction

For any technique to be more easily evaluated and applied to programs in the real-world, it must be automated. This chapter describes a suite of prototype tools that were developed to enable the rapid evolution and evaluation of the state-based testing technique.

Three tools were developed, MKTC (MaKe Test Cases), MKMFTC (MaKe a MakeFile for Test Cases), and TESTRUN. The tester creates a Test Control Script (TCS) in which is described the test cases that are to be generated. The TCS uses a simple language to describe the starting and ending states of test cases. Code for generating the states is provided in the TCS. MKTC uses the TCS to generate individual source programs for each of the test cases. MKMFTC is then used to generate a Makefile for automating the compilation of the test cases. TESTRUN is used to execute the test cases; the results are record along with any output the test cases generated. It is also possible to execute a command after each test case, this is provided so that coverage results can be accumulated.

The information in this chapter has been summarised from an earlier report [140].

Each tool will now be described in more detail.

7.2. MKTC

The MKTC tool was designed to automate a large proportion of the repetition involved in generating state-based test cases. The creation of the object under test's initial starting state, the validation of this state, and the validation of the correct state upon termination are automated. Information is supplied to the tool in the TCS. The information is described using the simple yet expressive language designed for the TCS.

MKTC can also generate an assertions class which allows state-based testing style assertions to be inserted into non-state-based test cases. Assertions are described using the same state-notation that forms part of the language used by the TCS. This enables the tester to write those test cases that are not easily expressed in a TCS.

A report can be generated, listing the details of all of the test cases generated. It contains information such as the starting state, finishing state, and the code used to create the starting state for each test case. It is also possible to generate a report listing all the **generators**[19] that could not be found. With this list, the tool can be used in two passes, the first to check the syntax and semantics of the TCS, in addition to listing the specification of all the generators that were missing (all of them at this point). The required generators could then be added to the TCS, and the tool run again to generate the test cases.

7.2.1. Options

The parameters that can be passed to MKTC are as follows:

/D - This option is used to turn debugging output on. The screen will then be filled with information about what MKTC is doing during the processing of the TCS. This is especially useful when there is a syntax error, as the parser provides a particularly verbose description of its current location within the file. It is also useful when validating that all the required test cases were generated. If, for example, the tester describes a test case which is invalid when tested against the invariant provided in the TCS, then this is not normally reported, but simply skipped; only if the test case declaration failed to

---

19 A generator is a tester-provided portion of code that is used to generate a particular starting state for test cases.

generate a single test case will an error be reported. The reason for this will become more apparent after the section which describes the declaration of the test cases within the TCS.

/A - This option causes MKTC to generate the assertions class - ASSERT. This class, as described earlier, is used for the inline checking of an object's state.

/J - This option causes MKTC to stop as soon as the parsing and basic semantic checking of the TCS file is over. It is useful for a simply checking the format and construction of the TCS without performing any further tasks such as the generation of test cases.

/N - This option causes MKTC to perform everything, except the actual generation of the test cases. This means that the tool goes through the process of generating the test cases, without actually writing them to a file. It is therefore possible to generate a report of the test cases that would have been produced, had the N option not been supplied. All the other appropriate options are still valid except /J.

/I - This option causes the regeneration of only those test cases that have changed since the last time that MKTC was called. It was incorporated for use with the Makefiles produced by MKMFTC. Only the changed test cases will be recompiled when used with a Makefile.

/F <filename> - This flag is not optional as it specifies the name of the TCS file that will be used for the generation of the test cases.

/R <filename> - This flag is optional and has two effects: it signals to MKTC to generate a report listing all of the test cases that are generated, and it supplies the filename into which the report will be placed.

/G <filename> - Similar to the R option, this one also has two effects: it signals to MKTC to generate a report of all the generators that it could not locate, it also specifies the name of the into which the report will be placed.

Of the above options, there are only two which are mutually exclusive, option J and option N. supplying both of these options will have the same effect as supplying option J alone.

The remainder of the options can be combined and supplied in any order, without affecting the execution of MKTC.

7.2.2. Files Used and Produced by MKTC

7.2.2.1. Test Control Script

This is the only user generated file that is required by MKTC. It is a plain ASCII text file which can be formatted by the user using both tab and control-characters if required. Comments may also be added into the file. These are simply discarded by the tool during the initial process of parsing.

The file contains seven separate sections, each of which starts with a unique keyword (except the first section). In the current version of the tool (version 3.0) the various sections of the TCS must be entered into the file in the prescribed order, failure to do so will result in an error from MKTC describing the section that was expecting next.

Each of the major sections starts with one or two keywords followed by a semicolon. It is then terminated by another keyword (END). However, this does not apply to the first section which describes general information about the testing that will be performed.

7.2.2.2. Report of Test Cases Generated

The first of the report files generated by MKTC lists details of the test cases that were generated. For each test case, the following information is provided:

- The test case number. This is useful for locating the test case because the test case number is part of the filename. If a test case failed, it can be debugged directly if required.
- The name of each object used in the test case. This is provided because of the ability to specify test cases that use more than one object from the class under test.
- The starting state of each object used. This is always described in state notation (see section 5.3.). If the starting state was originally described as a scenario, rather than a specific state, then this information is also supplied.
- The finishing state of each object used in the test case, this is also described in state notation.

- A list of the generators required to create the starting state for each object used in the test case.
- The name of the feature that is the centre of emphasis for the current test case.

If a test case declaration failed to generate any valid test cases it is mentioned in the report. This allows the user to trace which was the problem declaration from its position relative to the other test cases. This usually happens when the states used to describe the tests failed when checked against the invariant which was also declared in the TCS.

This file can also be used to validate the expected number of test cases, and to trace any differences that have occurred between the expected and the actual test cases generated.

7.2.2.3. Report of Missing Generators

A generator is a function provided by the tester for creating an object in a particular state. They are used to create the starting states for all test case. Generators are described in terms of their starting state (PRECONDITION[20]), their finishing state (POSTCONDITION), and the code required to change the object's state from the PRECONDITION to the POSTCONDITION. There is also an optional portion of code which is used to tidy up after a test case if required; its usual function is to destroy any objects created for the test case, that are not automatically destroyed.

When MKTC searches for a generator to be used for a particular test case it not only searches for a single generator, but it can also chain two generators together if necessary. At present the number of generators that can be chained together is restricted to two; however the search algorithm could be extended to search for a chain of any length in the future.

The report is simply a list of all the generators that could not be found, in the order that they were needed. As the test cases are generated, generators are reused over and over, therefore it is likely that a missing generator will be discovered more than once.

7.2.2.4. Test Cases

Test cases are generated one per file. Each test case has a unique number. In the current version of MKTC, the test cases are called STEST<number>.CPP, where <number> is the test case number

---

20 These words in capitals are the actual keywords that are used in the TCS.

between 1 and 999. The part of the test case that was actually generated by MKTC (rather than copied verbatim from the TCS) will be syntactically correct, however MKTC does not check the syntax of the code provided by the tester. This operation is left for the compilation stage.

The files containing any extra classes required for the correct compilation of the test cases can be specified in the header part of the TCS.

7.2.3. State Notation

The same state notation described earlier in section 5.4.3. is used in the TCS as a concise method for expressing that states involved in each test case. The notation has been expanded to include both scenarios and wildcards.

As mentioned before, a substate's value is specified by:

S[x](y)

where x is the substate number, and y is the chosen substate-value number. There are also symbols with special meanings which can be used in place of the y value:

- * - This symbol is expanded to mean all possible substate values. It is used in test case declarations to cause the generation of multiple test cases.
- ? - This symbol is used to match any substate value. It is usually used to express the finishing state of an object, by inferring that the value it currently holds does not matter.
- $ - This symbol is used to describe a change in the actual value of the substate, rather than a substate-value change. It is mainly used in conjunction with the negation operator ('!') to describe no change in value.

In addition to the above symbols, individual expressions can be negated, thus inverting its result. For example,

S[1](1) & !S[2]($)

declares substate one, value one, and no change in the value of substate two.

In addition, multiple substate-value numbers can be provided which will be expanded by MKTC into multiple test cases. For example

S[1](1,2..3,4.5..6)

In a similar manner to the symbol '*', the above list is used to express more than one state, thus generating multiple test cases, each of which will be validated against the supplied invariant.

7.2.4. Generation of the Test Control Script

This section briefly describes the various sections of the TCS along with the information that they contain.

7.2.4.1. Header Information

This section defines general information, the majority of which is not actually necessary for the generation of the test cases; it simply aids the tester in recognising and understanding the TCS files at a later date. The information is also required for the test generation report. It comprises of:

- the test engineer's name,
- the name of the class under test,
- the date the file was created,
- the Include files that are required in order for MKTC to create test cases that will compile correctly.

7.2.4.2. Substate Declarations

The next section of the TCS defines the substates that will be used during the generation of test cases. The whole of the state-based testing process and its effectiveness depends upon the careful choice of these substates.

The substates must be defined in ascending order, That is, 1, 2, 3, ..etc. Moreover, it is recommended that by each declaration the tester inserts comments describing the actual value, or values each substate-value corresponds to. This forms a useful reference guide whilst constructing the remainder of the TCS. These declarations enables MKTC to check the consistency of the remaining sections of the file.

7.2.4.3. Invariant Declaration

The next section of the TCS defines the invariant that will be used by MKTC to check the validity of test cases' start and finish states. In addition, the name of the operation that applies the invariant to the state of the object is specified here. Although MKTC has been designed for use with the C++ language, the current version does not accept all the different possible names for operations that are possible in C++.

7.2.4.4. Scenarios Declaration

This section of the TCS describes the data scenarios to be used during the test case generation. Each one is described by a state description. It is not necessary that the states be distinguishable from each other as their appropriate generators are accessed by specifying the scenario number.

7.2.4.5. Generators Declaration

The next section describes the generators that will be required by MKTC to create the starting states for all test cases. A generator is a segment of code which assumes that the object's current state matches its PRECONDITION in order for it to create a state which matches its POSTCONDITION. MKTC creates test cases for the generators before any others, thus if there is a error in a generator, it should be detected (assuming its failure is not intermittent) before any other test cases are executed.

The PRECONDITION can be either a normal state definition, or the special state - NONE. NONE declares that the Generator creates the object under test and therefore the object under test does not exist prior to the generator being called.

The POSTCONDITION can be either a normal state declaration, or a SCENARIO. If it is a SCENARIO, then the actual POSTCONDITION's state is taken from the SCENARIO's declaration earlier in the TCS.

The first segment of code supplied for the generator is the generator itself. If the PRECONDITION is NONE then it must contain a statement which dynamically allocates an object of the class under test to the variable "MKTC_Object".

The second segment (known as the terminator) is optional; its purpose is to tidy up anything left lying around by the generator. For example, dynamically allocated objects that need to be returned to the heap, or open files.

The current version of MKTC performs a two level search for a generator. That is, it searches for one generator with a PRECONDITION of NONE and a POSTCONDITION that matches the required state. If one cannot be found, a search is then made for two generators which when used one after the other, will create the required state. It may be possible to extend this search to an N level search, where N is the total number of generators defined for the object, however, at present this is not deemed to be necessary.

7.2.4.6. Declaration of Substate Testing Features

This section's purpose is to inform MKTC of the names of the features of the class under test that facilitate the testing of the substate-values. There is at least one feature required for each substate, with an optional second. The first simply allows MKTC to ask questions of the form:

"Is the substate value of substate x equal to y ?"

Where x is the allocated substate number, and y is the allocated substate-value number.

The second optional feature is one which is used in conjunction with the symbol '$' allowing MKTC to ask questions of the form:

"Has the actual value of substate x changed, since last asked ?"

where x is the allocated substate number.

7.2.4.7. Definition of Test Cases

This is the final and arguably, the most important section of the TCS. Its purpose is to describe to MKTC the test cases that should be generated. Being the largest and the most complicated section, it is also the most prone to human error.

The section is broken down into sub-sections, at least one per feature of the class under test. Each sub-section is made up of multiple <case_definition>s. A <case_definition> is a description of the state changes expected, and the code that should cause the state changes. From each <case_definition>, it is possible to generate multiple test cases, therefore no direct correlation exists between the number of <case_definition>s and the number of test cases generated.

It was felt that the more repetitions performed by the tester whilst generating the TCS, the more likely it would be that errors would be introduced into the file. Therefore, the notation used to express test cases is as concise and expressive as possible.

The state change that is expected to take place during the course of the test case, in its basic form describes the exact starting state and the exact finishing state. In its more complex form it can use the special symbols described earlier to generate multiple test cases from one description. It is also possible to provide multiple descriptions of state changes to be used with the same piece of code. Hence the tester has to write the minimum in test code.

Another feature provided by MKTC is the facility for using more than one object for each test case. This allows the testing of operations such as the assignment operator that use multiple objects of the same type. The states of each object can be described in terms of each other, hence an assignment operator can be described in terms of one object finishing in the same state that the other one started in.

The reason for the tester supplying the code, rather than it being automatically generated, is connected with the intricacies of constructing objects. The infinite possibilities that exist for the types of objects that are passed as arguments, means that it is far easier for the tester to simply insert a short piece of code to generate the required parameters; thus saving MKTC a great deal of work, which would otherwise over complicate the tool.

The portions of code that are provided by the tester for exercising the feature under test, that is applying a stimulus, are usually very simple pieces of code. Their purpose is two fold: they must create all the objects that will be passed as parameters to the feature; secondly, they must pass these arguments to the feature, and to compare any results that are returned. The comparison of results is optional, as these tests will be performed by the functional tests that should also be performed.

The only major assumption that is made by the tool is concerned with the validity of the extra features added to the class under test. It assumes that the routines for testing the values of the substates of the object under test are error free. These routines are usually very simple, consisting of a list of tests which determine the current substate-value. An error in these routines would be particularly difficult to check as it is likely that test cases would fail in a seemingly random order.

7.2.5. The Generation of the Test Cases

MKTC first parses the whole of the TCS file, storing all the information in a compressed format in memory. The syntax and semantics checking is mostly performed as the file is parsed. However, certain details such as reference states cannot be resolved until the test case definitions are expanded.

Each test case declaration is expanded. The starting state of the test case is checked against the invariant so that there are less invalid test cases generated. Also the finishing state is checked. If both states are valid, MKTC then searches the list of generators for a series of generators that can create the required starting state. The details of the test case are then added to the report (if the report is being generated) and the test case is then created.

7.2.6. The Assert Module

Assert is designed to provide the tester with the ability to directly insert statements to check for specific states at run-time.

Assert is a class which allows the user access to a uniform approach to the testing of assertions within a test case. The tester can either test Boolean values, or state descriptions. Both types of tests cause a simple 'yes'/'no' answer. Any value can be converted to a Boolean by simply comparing it with a value of the same type; for example, 10 = 15 would simply result in false.

State definitions are passed to test() as a string. For example,

```
assertion.test(string("s[1](1) & S[3](4) & !S[4](2);"));
```

The assert module exports a global object called *assertion*. It is provide to save the tester from creating a new object every time an assertion is required.

7.3. MKMFTC

7.3.1. Introduction

The next tool to be described is the tool MKMFTC. When using MKTC, the tester will have a directory containing a large number of test cases in source format. Each test case will require

compiling and linking with the correct libraries, ready for execution. This is an extremely repetitive process and an obvious candidate for automation.

Automatic compilation and linking is achieved with the Make[21] tool, along with a custom Makefile. A Makefile (in its simplest form) describes which source files, and commands are needed to be to create the required executable. A Makefile is a series of rules; each rule (in general) describes the target, the files that the target is dependent upon, and how to get from the dependencies to the target. The target can be either a file, or a dummy target. A dummy target is used to allow the user to specify groups of files to be made, for example, the dummy target 'all' is conventionally used to mean 'make all of the targets described in the Makefile'.

The basic top-level rule defined by MKMFTC is 'all'. Therefore the command 'make /f<makefile name> all' would cause Make to compile and link all of the test cases. This result is achieved by making the dummy target 'all' dependent upon all of the executable test cases. Each of the executable test cases is dependent upon the particular test case's object file, and the object file is dependent upon the source file.

MKMFTC takes a small file in the current directory which expressed the dependencies between the test cases and any other modules required. This file is combined with information obtained from a listing of the current directory to form the basis of the rules generated.

There are two basic categories of test cases that can be manipulated by MKMFTC. These are : state-based tests, and any other test cases. State-based test cases have been described earlier. The other test cases are any others generated by the tester. These usually include both functional and structural tests.

State-based test cases are expected to have filenames of the format,

stest<number>.cpp

and the other test cases are expected to have filenames of the format,

test<number>.cpp

---

21 Make is a utility that is provided with the vast majority of development environments. At present MKMFTC has been designed to use the Make utility provided with Borland C++ 3.1, however it is a relatively simple task to adapt it for use with other versions of Make.

The two types of test cases are handled separately, with each type having its own Makefile. Both Makefiles can be generated by MKMFTC. The Makefiles are called,

stesting.mak - state-based test cases

testing.mak - other test cases

7.3.2. Parameters

The options that may be supplied to MKMFTC as arguments on the command line are listed below:

/S [<start number> [<end number>]]

This option causes a Makefile to be created for the state based test cases in the current directory. Optional start and end numbers may be supplied in which case the Makefile will only contain details for the test cases whose number falls between the supplied start and end number. Also, the end number is option, allowing all test cases from the start number to the last test case.

/O [<start number> [<end number>]]

This option is similar to /S in operation, but causes a Makefile to be produced for the non state-based tests.

/D

This option causes the program output to be more verbose. The user is informed exactly what MKMFTC is doing during the generation of the Makefile(s).

/?

This option causes the available options along with a brief description to be displayed on the screen.

7.4. Automatic Execution of Test Cases (TESTRUN)

7.4.1. Introduction

The final tool to be discussed is TESTRUN. TESTRUN is designed to automate the execution of large numbers of test cases. In a similar manner to MKMFTC, TESTRUN can manipulate both state-based and non state-based test cases.

TESTRUN is capable of generating two output files: a file containing all the output from the test cases (the results file), and a file which records the filenames of the test cases that failed. In the results file, each test case is preceded with a line giving the test cases' number. This ensures that test cases can be easily located using the simple search facilities that are usually provided with most text editors. The word 'FAILED' is used to describe the test cases that have indeed failed. This is also a useful tag to search for using a text editor.

The file listing the failed test cases is very terse. It simply contains a list of those test cases that failed. The test cases that fail are also reported to the screen, but screen output is transient by nature, thus it is recommended that this output file also be generated.

7.4.2. Parameters

A number of options are available which affect the functionality of TESTRUN. These are:

/S [<start number> [<finish number>]]

This option causes TESTRUN to execute the state-based test cases found in the current directory. If the start number is specified, all the test cases from start number onwards will be executed. If the finish number is also specified, all the test cases between the start number and the finish number are executed.

/O [<start number> [<finish number>]]

This option is very similar to /S in function except that it executes the non state-based test cases.

/F <filename>

This option forces TESTRUN to store the output from all of the test cases in the filename specified. This file is useful for debugging the test cases that failed.

/E <filename>

This option causes TESTRUN to record in the filename provided, the names of any test cases that fail.

/C <command sequence>

The command sequence specifies a command that will be executed after each valid test case. This was designed to be used in conjunction with a coverage analyser. Hence new coverage statistics could be analysed after each test case.

7.5. Summary

This section has outlined three prototype tools which can be used to generate and execute a state-based test suite. The TCS file used by MKTC is concise yet powerful. The current version requires that the tester writes short pieces of code for the generators and for the test cases themselves. It is possible that in future versions, the tester will be required to write far less code. Also, test cases are stored in their own individual source files, when compiled, this can lead to a requirement for a large amount of disk space. It is hoped that this will be addressed in a future version of the tool.

MKMFTC is used to generate a Makefile for the automatic compilation and linking of all test cases stored in the current directory. The use of a Makefile has the added bonus of only recompiling those test cases that have changed. Hence if used in conjunction with the correct option to MKTC, the time taken to regenerate the test cases is drastically reduced.

TESTRUN allows multiple test cases to be executed and monitored easily. It can also be used to aid in the gathering of coverage statistics.

Although only prototype tools, they are all effective at generating and executing a state-based test suite,

Chapter 8.

Case Studies

8.1. Introduction

In chapter 5. the process of state-based testing was described in detail. This chapter describes four case studies where state-based testing has been applied to real code. The classes are: a linked list class for storing integers, a class for providing and manipulating strings of text, simple lexical analyser, and a more complex lexical analyser. A suite of state-based test cases were generated, and the code coverage[22] measured when tests were applied to the classes. From the coverage, the unexecuted paths were determined. This information has been used in conjunction with the source code to determine the precise situation when the paths would be executed.

The case studies and the coverage measurements will be used in the next chapter to evaluate the effectiveness of state-based testing for finding errors and achieving high levels of executed code coverage.

The remainder of this chapter is structured as follows: the first section describes the main objectives of the case studies. This includes an outline of the structure of the case studies. Following the objectives

---

22 The code coverage of a suite of test cases is depependent upon the type of coverage measure chosen. This will be discussed in more detail later.

is a detailed description of the method undertaken for each case study. Included in this section is a discussion of the metric used for the code coverage measurements. Finally, there is one section for each of the four case studies. No summary of this chapter is provided. It will be included in the next chapter's summary (the evaluation chapter).

8.2. Objectives

The purpose of this chapter is to provide some data and statistics that can be used to evaluate the state-based testing technique. Each case study class is structured as follows: The design and use of the class is briefly outlines along with any general design information that is required to generate a complete state-based test suite. Next, is a description of the substates and scenarios chosen for each class. These are determined by following the algorithms outlined in chapter 5. A Test Control Script (TCS) is generated for each class; these can be found in the appendices.

The code coverage of the class is broken down into the coverage for each operation. This information is used to determine which sub-paths through each operation were not executed. Each unexecuted sub-path is analysed to determine whether it is infeasible or not. An infeasible sub-path is one for which there is no combination of values for variables that would result in its execution. The knowledge of the infeasible paths is used to recalculate the code coverage for the class. The recalculated figure is a more accurate reflection of the executed paths compared to the unexecuted paths. The factors affecting the coverage achieved with a state-based test suite will be discussed in the next chapter. However, a brief discussion of the results of each case study is provided at the end of each case study.

As with the other examples in this thesis, the source code is written in the C++ language. Therefore when discussing the intricacies of each class it may be necessary to use symbols and concepts from the language.

8.3. Method

Each class is tested using the tools described in the previous chapter. This entails creating a Test Control Script (TCS) for each class which describes the states of an object under test and the test cases that should be generated. The test cases are then compiled and executed. Any failed test cases are carefully checked to determine whether an error exist in the description of the test case, or the class

under test contains an error. Corrections are made to the error and the failed test cases regenerated. This process is repeated until all test cases execute correctly.

The coverage of the suite of test cases is measured. The statistics will be used in the discussion of the results to demonstrate which portions of a class can and cannot be validated by state-based testing (see next chapter). The coverage technique chosen for the evaluation is LCSAJ (Linear Code Sequence and Jump) coverage. An LCSAJ is a sub-path through an operation; it terminates when the next statement to be executed is not the next consecutive statement in the program. For a more detailed description of how the LCSAJs are determined for a specific source program, see [137, 155, 59].

LCSAJs were chosen as the coverage measure because they provide a greater degree of test coverage than branch coverage. They also require that loops are executed once, twice, and multiple times.

The LCSAJ coverage of the test suite is calculated by instrumenting the code under test with markers at strategic points. A marker is a function call to a routine that outputs a unique number to a markers file. The unique number identifies which particular portion of the code is currently under execution. After the test case has terminated, the sequence of markers in the markers file is analysed to determine which LCSAJs have been executed. The tools used were adapted from a suite of coverage tools for the Modula-2 language. The Modula-2 tools and the positions for the insertions of the markers are described in [137]. The insertion of markers into the code was performed by hand due to unavailability of a suitable automated tool for the C++ language.

The LCSAJ coverage of a program is defined by the Test Effectiveness Ratios (TERs). They are defined as follows [155]:

$$TER_{n+2} = \frac{\text{number of distinct sub - paths of length n or less LCSAJ's exercised}}{\text{total number of distinct sub - paths of length n or less LCSAJ's}}$$

For the purpose of this evaluation, TER_3 has been chosen as the metric, therefore, all individual LCSAJ's must be executed at least once.

A major concern when using LCSAJs as a coverage measure is the infeasability (unexecutability) of certain paths through the program. Their effect on the testing process is that 100% LCSAJ coverage can rarely be attained. The main causes of infeasible paths are defensive programming and the use of control-flow termination constructs such as 'return', within conditional constructs. For more information on infeasible paths, see [56].

8.4. Case Study 1 : a Linked List of Integers

The first class that will be used for the evaluation is a linked list of integers - ilist. It is based on a linked list template written during the construction of the tools described in the previous chapter. It has been manually instantiated to form a linked list of integers for the purpose of testing, as templates cannot be directly tested. A listing of the class is provided in appendix 1.

8.4.1. Background

The class ilist has two attributes: pTop and pCur. pTop is a pointer to the first node of the linked list, whereas pCur is used to refer to the current item in the list. pCur allows the programmer to 'walk' from the start to the end of the linked list one node at a time. It is implemented as a pointer to a pointer to the current node, so that even when the list is empty, pCur points to a valid pointer (that is, pTop).

When pTop is NULL, then the list is empty and pCur points to pTop, that is, pCur = &pTop. pCur is never NULL even if the list is empty. This simplifies the insertion routine. Nodes can either be added to the end, or the beginning of the list, or they can be inserted at the current place in the list.

The operations of the class are summarised below:

CDT_Slist() - (The default constructor), its purpose is to initialise the attributes of the class to represent an empty list.

CDT_Slist(...) - (The copy constructor), this is used to create a copy of a list. It creates an identical list with the elements in the same order.

~CDT_Slist() - (The destructor), this is used to return any memory allocated to the linked list (that is, the nodes) to the system's memory heap.

clear(...) - This operation empties the list of its elements, and returns any memory allocated, back to the system's memory heap.

prepend(...) - This operation inserts a new node at the top of the linked list.

append(...) - This operation inserts a new node at the bottom of the linked list. to achieve this, it must first traverse the whole list to reach the bottom.

find(...) - This operation searches the linked list for a specific element. It returns true if the element was found in the list, otherwise it returns false. It uses the equality operation that must be provided by the element.

remove(int ...) - This operation searches through the linked list for a specific element, if it finds the element, the element is removed from the lists and the operation returns true. Otherwise the operation returns false.

replace(...) - This operation searches through the list for a specific element, if it is found, the element is replaced by a specified one. The operation returns true if the element existed and was successfully replaced, otherwise it returns false.

first() - Initialise the pointer to the current node to the head of the list. Return true if the list is not empty, otherwise return false.

next() - Move on to the next element in the list. Return true if there is a next element, otherwise return false.

get(...) - Get the value of the current element in the list. If there is no current element, return false.

set(...) - Set the value of the current element in the list. If there is no current element, return false.

insert(...) - Insert a new element in the list before the current element. If no current element exists, insert it at the head of the list.

remove(...) - Remove the current element in the list. Return false if there was no current element.

toend() - Count the number of elements from the current element to the end of the list.

size() - Return the length of the list.

operator=(...) - Copy the contents of one list to another, creating two identical lists.

operator!=(...) - compare two lists for inequality.

operator==(...) - compare two lists for equality.

A structure (Slink) is used to implement the nodes of the list. Slink stores the element value and a pointer to the next node.

8.4.2. The Class's Substates

The data -representation of the class is:

```
Slink * pTop; (a pointer to a node)

Slink ** pCur; (a pointer to a pointer to a node)
```

From this and the design information (not shown), the following substates and substate-values were chosen:

For S[1]:

1) pTop equals NULL (a specific-substate-value)
2) pTop is not equal to NULL (a general-substate-value)

For S[2]:

1) *pCur equals NULL (a specific-substate-value)
2) *pCur is not equal to NULL (a general-substate-value)

For S[3]:

1) pCur equals the address of pTop (a specific-substate-value)
2) pCur is not equal to the address of pTop (a general-substate-value)

Three additional operations were added to the ilist class to facilitate the testing of the above substate values. Another three additional operations were added to detect a change in value of the substates.

8.4.3. The Class's Scenarios

The ilist class was previously used as an example to during the description of scenarios in chapter 5.6.1. The following six scenarios are a summary of the information presented earlier:

1. an empty list
2. a single element list with the current pointer at (before) the element
3. a single element list with the current pointer after the element
4. a multi-element list with the current pointer at (before) the first element
5. a multi-element list with the current pointer at (before) the last element
6. a multi-element list with the current pointer at (before) an element in the middle of the list

These are used to generate the state-based test suite for this class. The complete Test Control Script is provided in appendix 2

8.4.4. The Class's Additional Substates

To aid the use of the above scenarios and to detect changes in the current state of the dynamic data structure (the linked list) the following additional substates have been chosen:

S[4]to detect `(*pCur)->plNext`

1) `(*pCur)->plNext` equals `NULL`
2) `(*pCur)->plNext` does not equals `NULL`
3) invalid (`*pCur` equals `NULL` and so cannot be dereferenced)

S[5]to detect `(*pCur)->item()`

1) `(*pCur)->item()` is valid
2) `(*pCur)->item()` is invalid (`*pCur` equals `NULL` and so cannot be dereferenced)

S[4] is used to detect whether or not the current pointer is pointing at the last node in the list. S[5] is provided not for the direct detection of substate-values, but for the detection any change in the value stored at the current node in the list.

An extra attribute has been added to the class so that when it is set to true, it informs the operations of the class that they are to simulate an out of memory fault. This attribute, when combined with state-based testing is used to determine that although an exceptional circumstance has occurred, objects of the class are left in a valid state.

8.4.5. Test Coverage Results

Below, the test coverage results obtained after the execution of the complete suite of state-based test cases are listed. The individual coverage of each feature is reported, along with the total coverage of the class. This is followed by a discussion of the unexecuted LCSAJs. In addition, explanations of the conditions needed for their subsequent execution are provided. Finally, a more accurate coverage measure with the infeasible (unexecutable) LCSAJs taken into account is provided.

The coverage level achieved after the state-based testing was: 87.16%, that is 95 LCSAJs were executed out a total 109 LCSAJs.

The coverage results for individual methods were as follows:

CDT_Slist(void)	=	100.00%	clear(...)	=	100.00%
CDT_Slist(...)	=	100.00%	prepend(...)	=	100.00%
~CDT_Slist()	=	100.00%	append(...)	=	90.91%

find(...)	=	80.00%	insert(...)	=	100.00%
remove(int ..)	=	71.43%	remove(...)	=	100.00%
replace(...)	=	70.00%	toend()	=	62.50%
first()	=	100.00%	size()	=	100.00%
next()	=	100.00%	operator=(...)	=	100.00%
get(...)	=	100.00%	operator!=(...)	=	100.00%
set(...)	=	100.00%	operator==(...)	=	87.50%

The LCSAJs not executed during the state based testing are as follows:

append(...) :

- The LCSAJ between line 89 and line 91. This one runs from the start, to the end of the `while` loop. Execution would occur if the length of the list is greater than 3 elements.

find(...) :

- The LCSAJ beginning and ending on line 108. This one starts at the end of the 'if-then-else-endif' construct. It finishes at the end of the while loop. It would only be executed if the 'if-condition' evaluated to TRUE. However, that branch has a return statement, therefore rendering this LCSAJ infeasible.
- The LCSAJ between line 108 and line 109. This one starts at the end of the 'if-then-else-endif' construct. It ends at the return statement. Its execution is infeasible for the same reason as the previous LCSAJ above.

remove(...) :

- The LCSAJ between line 113 and line 121. This one is also infeasible because expression `(pLast == NULL)` always evaluates to TRUE when the while loop is first entered.
- The LCSAJ between line 117 and line 127. This one is also infeasible because `(pLast == NULL)` always evaluates to FALSE after one iteration of the while loop.
- The LCSAJ between line 139 and line 140. This one is also infeasible because of the `return` statement in the previous branch of the 'if-then-else-endif' construct (line 133).
- The LCSAJ between line 139 and line 141. This one is also infeasible for the same reason as the previous LCSAJ above.

replace(...) :

- The LCSAJ between line 148 and line 150. This one would be executed by a valid replace performed on any element other than the first or the second of a list.

- The LCSAJ between line 157 and line 157. This one is infeasible because of a `return` statement in the previous branch of the 'if-then-else-endif' construct (line 153).
- The LCSAJ between line 157 and line 158. This one is also infeasible for the same reason as the previous LCSAJ above.

toend(void) :

- The LCSAJ between line 251 and line 255. This one starts at the end of the 'if-then-endif' construct. It finishes at the start of the `while` loop. It is infeasible to execute because the condition of the `if` and the condition of the `while` are mutually exclusive, therefore both would not evaluate to FALSE at the same time.
- The LCSAJ between line 255 and line 259. This one would be executed when this function is performed with a distance of greater than one element to the end of the list from the current place.
- The LCSAJ between line 259 and line 260. This one is infeasible because it requires the `while` condition to evaluate to FALSE before its first iteration, however, the `if` on line 249 would already have caused a return from the function.

operator==(...) :

- The LCSAJ between line 273 and line 274. This one would be executed when testing the equality of two objects, A and B, when B contains all of A's elements and more. That is to say, that A has a subset of B's elements.

With the infeasible LCSAJs (10) taken into account, a more accurate coverage of the class is 95 from a feasible 99 LCSAJs, that is, 95.96% coverage.

8.4.6. Discussion of Results

Using the state-based technique, 203 test cases were generated using a TCS and the tools MKTC and MKMFTC. The first six test cases only test the generators described in the TCS, ensuring that they are indeed valid to be used for the remainder of the test cases. Tests are also inserted into each test case to validate that the test cases starts in its expected starting state.

As can be seen from the coverage results above, every operation has been executed by a number of different test cases; in some situations this caused 100% coverage of the operation. However, some of the more complex operations still require further testing with functional and structural techniques to achieve the desired coverage level of 100% feasible LCSAJs.

Overall, a very high coverage level has been achieved with the state-based test cases alone (95.96%). This can be attributed to the concentration of the class's functionality for the manipulation of substates and their values. Minimal manipulation of the parameters passed to operations occurs. Also, the code's simplicity increases the ease with which it can be exercised by simple test cases. The class - ilist, uses a high level of control-information within the representation of the class in contrast with a low level of complexity within each feature.

An example of an error that might be found in a class of this type would be if upon termination of a call to clear( ), S[3] was not reset to S[3](1), that is, the pointer to the current element (pCur) was not reset to &pTop after all the nodes of the list had been destroyed. The effect of this error would not be easily detectable via the interface of the class, even though pCur was left pointing to a block of memory which no-longer belongs to the linked list. This could have disastrous results if operations such as `next( )` are then called. This example error was one which was actually found in the code during its validation. A more detailed discussion of the types of errors that can be detected by state-based testing is provided in the next chapter.

8.5. Case Study 2 : a String Class

The second class to be used as a case study is a string class providing control, storage and manipulation of text strings. A listing of the class is provided in appendix 3.

8.5.1. Background

The string class provides an encapsulation for the standard strings that are used in the C and C++ languages. Traditionally, strings are stored in an allocated block of memory with a trailing null character[23] to mark the end. With standard strings, the size of the block of memory used for storage is not stored for reference, hence it is conceivable that an operation might overrun the end of the allocated block. System routines have no method for checking if a block of memory is big enough for an operation, they have no choice but to assume it. This can lead to some operations having undefined results on the contents of a programs memory if not enough memory was allocated.

---

23 A character whose ordinal value is zero.

The string class 'remembers' the size of the allocated block. If the text resulting from an operation is too long for the memory block, a new block large enough to accommodate the text is allocated. However, if the size of the memory block is larger than the text, a smaller block is not allocated, as this would fragment the systems memory into smaller and smaller blocks.

An extra character is allocated at the end of the memory block into which a '\0' is stored. This is so that there will always be a string terminator for the text whether the programmer has provided one or not. As the null character is used to detect the end of the string, operations will no longer inadvertently 'wander' past the end of the text when searching for the end.

If no block of memory is allocated, that is, the string is in a NULL state, the string uses a character containing '\0' (an empty string) to avoid problems of trying to use a NULL string. Hence, if a string object is converted to a standard string (by returning the pointer to the memory block), the conversion always returns a valid result, even if it is only an empty string.

Operations are provided to manipulate the strings, allowing their combination with other strings. All required memory allocation is handled automatically.

8.5.2. The Class's Substates

The representation of the class is:

```
int iSize;

char cEnd;

char * sptr;
```

From these, the following substates were chosen:

For S[1]:

1) iSize equals zero (a specific-substate-value)
2) iSize is greater than zero (a general-substate-value)

For S[2]:

1) sptr equals NULL (a specific-substate-value)
2) sptr equals the address of cEnd (a specific-substate-value)
3) sptr points to a valid memory block (a general-substate-value)

Substate one reflects the size of allocated block of memory. when iSize equals zero, there is no memory block.

Substate two reflects whether a valid memory block exists or not.

There is no substate for cEnd, as its value should never be changed. It is a dummy variable used to store and end of string marker when there is no memory allocated to the string. The invariant for the class will fail if a change has been made to the value of this attribute.

8.5.3. The Class's Scenarios

The class has a number of situations which should cause a new block of memory to be allocated. There are also situations when a new block of memory should not be allocated. The following five scenarios were chosen so that these situations could be tested with the aid of substate S[2], the scenarios are:

1) an empty string,
2) a string with a small memory block,
3) a string with a large memory block,
4) a string with text right up to the end of the memory block,
5) a string with text which does not reach the end of the memory block.

For example, assigning an object from scenario 4 to an object currently in scenario 1 would cause a new memory block to be allocated. Whereas assigning an object from scenario 1 to an object of any scenario from 2 through to 5 would not cause a new memory block to be allocated.

No additional substates are required to attain maximum benefit from the above scenarios. However, an extra attribute has been added to the class so that when it is set to true, it informs the operations of the class that they are to simulate an out of memory fault. This attribute, when combined with state-based testing is used to determine that although an exceptional circumstance has occurred, objects of the class are left in a valid state.

The operations of the class are summarised below:

create( )	-	an internal operation for allocating a block of memory for storing the strings in.
destroy( )	-	an internal operation for releasing the memory that has been previously used for storing strings.
String( )	-	the default constructor for initialising an empty string.
String(const String &)	-	the copy constructor for creating another string identical to the one passed as a parameter.

String(int) - an additional constructor for creating an empty string which has memory allocated for storing a string of the length specified.

string(char *) - another additional constructor for creating a new string identical to the standard string passed as a parameter.

~string( ) - the destructor, which ensures that all memory is correctly freed upon termination of the objects life.

operator=(..) - copies the text from one string object to another.

operator+=(..) - concatenate one string on to the end of another.

operator+(..) - concatenate one string onto the end of another and return the result in a new string object.

operator==(..) - compare two strings for equality.

operator!=(..) - compare two strings for inequality.

operator<(..) - compare the current string to with another string to see if the other string would alphabetically precede the current string. operators >, <= and >= also perform similar functions.

find(..) - search forwards backwards from a chosen point on the string for a specified character.

left(..) - return the specified number of characters from the left hand end of the string.

right(..) - return the specified number of characters from the right hand end of the string.

mid(..) - return the specified number of characters starting at a specified point in the string.

tolower( ) - convert all the characters in the string to their lower case equivalents.

toupper() - convert all the characters in the string to their upper case equivalents.

length( ) - calculate the length of the string.

storesize( ) - return the amount of space that can currently be occupied by the string.

empty( ) - test the string to see if it contains no characters.

operator const char * ( ) - convert the string object into a standard string.

operator[ ] - set the character at the specified position in the string.

operator[ ] const - get the character at the specified position in the string.

8.5.4. Test Coverage Results

The TCS for the string class was used to generate 452 test cases. The first 5 test cases tested the validity of the five scenario generators for use in the other 447 test cases. The scenario generators were used to generate all the initial states for the test cases. The test case results were validated by testing the final values of the substates.

The LCSAJ coverage of the string class after the test cases was 91.09%, that is 92 out of a total 101 LCSAJs were executed. The coverage values for the individual operations are listed below:

create(...)	= 100.00%	mid(...)	= 100.00%
destroy(...)	= 100.00%	~string( )	= 100.00%
string(void)	= 100.00%	operator==( )	= 100.00%
string(const string &)	= 100.00%	operator!=(...)	= 100.00%
string(int)	= 100.00%	operator<(...)	= 100.00%
string(char*)	= 71.43%	operator>(...)	= 100.00%
operator=(...)	= 76.92%	operator<=(...)	= 100.00%
operator+=(...)	= 81.82%	operator>=(...)	= 100.00%
operator+(...)	= 100.00%	length( )	= 100.00%
find(...)	= 85.71%	storesize( )	= 100.00%
left(...)	= 100.00%	operator const char * ( )	= 100.00%
right(...)	= 100.00%	empty( )	= 100.00%
tolower(...)	= 100.00%	operator[] const	= 100.00%
toupper(...)	= 100.00%	operator[]	= 100.00%

The LCSAJs not executed during the state-based testing are as follows:

*string(char *)*

- The LCSAJ between line 102 and line 106, and the LCSAJ between line 118 and 123 would be executed if the constructor was passed and empty string ("").

operator=(...)

- The LCSAJ between line 138 and line 143 is potentially infeasible because it requires iLen to be zero which would mean that the first branch of the if would not have been taken, because iSize is never less than zero.
- The LCSAJ between line 141 and line 143 is infeasible because of the return statement in the first branch of the preceding if.
- Likewise, the LCSAJ between line 141 and line 147 is also infeasible.

operator+=(...)

- The LCSAJ between line 175 and line 179 is infeasible because the execution of the first branch of the if results in bNew being set to true, hence the if predicate on line 179 must evaluate to true.
- The LCSAJ between line 172 and line 181 is also infeasible because execution of the else clause results in the if predicate on line 179 evaluating to false.

find(...)

- The LCSAJ between line 211 and line 215 would be executed if find was requested to search forwards and the specified starting point for the search was past the end of the string.
- The LCSAJ between line 208 and line 215 would be executed if find was requested to search backwards and the specified starting point for the search was past the end of the string.

Taking the infeasible LCSAJs into account, a more accurate measure of the coverage is 92 from a possible 96 feasible LCSAJs, that is, 95.83% coverage.

8.5.5. Discussion of Test Results

A high degree of coverage was also reached for string class. This is due to the simplicity of operations of the class, and the scenarios facility which enhances the basic state-based test suite. Apart from the 5 test cases which validate the generators described in the TCS, the remainder of the test cases all used the scenarios as their basis. The advantage of scenarios is that they provide a more specific situation for an object than can be described with the basic state notation, although the state notation is used to validate the results of the test cases.

Another factor affecting the high coverage of this class is the lack of manipulation of the parameters passed. This implies that the vast majority of the code for each operation is solely concerned with the manipulation of the object's state. Hence a suite of state-based test cases will cause a high level of execution coverage.

8.6. Case Study 3 : a Simple Lexical Analyser

8.6.1. Background

SimpleLex is part of a three class hierarchy for providing varying degrees of lexical analysis on an input stream. The input stream can be taken either from a file, or from a list of strings. Using the list of strings as an input stream allows test cases to be generated easily without the requirement for large numbers of sample files. The SimpleLex uses a standard interface to interact with the input streams, hence there should be no visible difference between either type of input stream.

SimpleLex inherits some of its functionality from its parent class BasicLex. BasicLex provides simple character by character analysis. SimpleLex uses the character analysis to analyse simple multi-character tokens such as numbers and words.

For the purpose of this testing, it is assumed that BasicLex has been validated using both state-based testing and traditional black and white-box testing techniques.

The operations of the class are as follows:

SimpleLex(..)	- the default (and only) constructor. It initialises all new SimpleLex objects to a known state.
~SimpleLex( )	- the destructor for tidying up at the end of an objects lifetime.
current( )	- get the last token that was analysed.
next( )	- analyse another token from the input stream.
Text(...)	- get the text of the current token.
MoveToNextLine( )	- skip the remaining tokens on the current line in the input stream.
skipto(...)	- skip the tokens in the input stream until one matches the set passed.

A listing of the class is provided in appendix 4.

8.6.2. The Class's Substates

The representation of the parent class BasicLex is:

```
BasicLex::token tkThis;     // the last character token analysed
int iCurPos;    // the current position in the current line of text
StreamOfTokens * sotData;  // The input stream of tokens
```

The representation of the class is:

```
SimpleLex::token tkThis;    // the last token analysed
int iStartPos;// the starting position of the token
```

SimpleLex inherits both the attributes and the functionality of BasicLex, hence the substates used during the state-based testing of BasicLex may also be required for the state-based testing of SimpleLex. However, if SimpleLex does not access the inherited attributes directly, they will not be needed.

The substates required from its parent class are:

For S[1]: this is a pointer to an input stream object

1) BasicLex::sotData equals NULL (a specific-substate-value)
2) BasicLex::sotData does not equal NULL (a general-substate-value)

For S[2]: This detects the end of the input stream

1) invalid - this is required because sotData may equal NULL
2) BasicLex::sotData->EndOfTokens() equals false (a specific-substate-value)
3) BasicLex::sotData->EndOfTokens() equals true (a specific-substate-value)

For S[3]: this is the lexical value of the last character analysed by BasicLex

1) BasicLex::tkThis equals Error (a specific-substate-value)
2) BasicLex::tkThis equals Unknown (a specific-substate-value)
3) BasicLex::tkThis equals EndOfFile (a specific-substate-value)
4) BasicLex::tkThis equals EndOfLine (a specific-substate-value)
5) BasicLex::tkThis equals any other value (a general-substate-value)

For S[4]: the current position in the input stream

1) iCurPos is less than zero (a general-substate-value)
2) iCurPos is greater than or equal to zero (a general-substate-value)

The substates for the attributes of SimpleLex are:

For S[5]: the lexical value of the last symbol analysed by SimpleLex

1) tkThis equals Identifier (a specific-substate-value)
2) tkThis equals Number (a specific-substate-value)
3) tkThis equals Symbol (a specific-substate-value)
4) tkThis equals EndOfLine (a specific-substate-value)
5) tkThis equals EndOfFile (a specific-substate-value)
6) tkThis equals WhiteSpace (a specific-substate-value)
7) tkThis equals Error (a specific-substate-value)
8) tkThis equals Unknown (a specific-substate-value)

For S[6]: The starting position in the input stream of the last symbol analysed

1) iStartPos equals zero (a specific-substate-value)
2) iStartPos is greater than zero (a general-substate-value)

8.6.3. The Class's Scenarios

Eleven scenarios were chosen to enhance the state-based testing. Of the scenarios, one is concerned with no input stream, three with the correct detection and response to of the end of the input stream, and the remainder with the correct detection of tokens.

The scenarios are:

1) NULL was passed as the pointer to an input stream, hence no input exists,
2) The end of the stream is a long way off,
3) The end of the stream is next, but has not been detected yet,
4) The end of the stream has been detected,
5) The next character to be analysed is an BasicLex::Alpha (that is, the start of an Identifier),
6) The next character to be analysed is a BasicLex::Digit (that is, the start of a Number),

scenarios 7 through to 11 are the same as scenario 6 but for the tokens BasicLex::Tab, BasicLex::Space, BasicLex::ControlChar, BasicLex::Symbol and BasicLex::EndOfLine respectively.

No additional substates were required.

8.6.4. Test Coverage Results

The LCSAJ coverage of the simple lexical analyser class is 87.76%, that is, 43 executed LCSAJs out of a possible 49. The coverage measures for the individual operations are:

SimpleLex(void)	= 100.00%	Text(...)	= 60.00%
~SimpleLex( )	= 100.00%	MoveToNextLine( )	= 100.00%
current( )	= 100.00%	skipto(...)	= 77.78%
next( )	= 92.00%		

An analysis of the unexecuted paths follows:

next( )

- The LCSAJ between line 51 and line 61 would be executed if next( ) was called and it encountered a single tab character which is followed by a character which is not whitespace[24].
- The LCSAJ between line 53 and line 61 would be executed if next( ) was called and it encountered a single control character which is followed by a character which is not whitespace.

Text(...)

- The LCSAJs between line 91 and line 93, and line 95 and line 96, Are due to defensive programming. The offending if statement ensures that we do not pass an illegal value to an operation of the class string.

skipto(...)

- The LCSAJs between line 114 and line 115, and line 118 and line 119 would be executed if the input stream was already at the End Of File, or the current token is the one being searched for.

There are no infeasible LCSAJs within the operations of the class.

---

24 Whitespace characters are normally considered to be those characters for which there is no visible ASCII representation. This includes spaces, tab characters and control characters.

8.6.5. Discussion of Test Results

State-based testing achieves a high level of coverage with this class. This can be attributed to low number of members of the enumerated type SimpleLex::Token. This has the effect of keeping the number of substate-values for each substate low. Consequently, it simplifies the state-based testing by reducing the number of generators that are required.

SimpleLex relies heavily upon the class BasicLex to provide its functionality. Hence, its complexity is low in comparison to a class which doesn't inherit functionality.

8.7. Case Study 4 : A More Complex Lexical Analyser

8.7.1. Background

The Lex class is the fourth class in the lexical analysis hierarchy. It is a descendant of the Simplelex class (described above).

It provides all the lexical analysis services that are required by the tool MKTC (described earlier). It inherits its general functionality from SimpleLex, however it analyses more complex tokens and hence is a more complex class. It is capable of analysing specific identifiers, and specific symbols, along with the more general identifiers and lexical tokens required by MKTC. It is capable of parsing any of the tokens and symbols that can be used to construct the Test Control Scripts.

Lex uses the lexical tokens recognised by SimpleLex as basic building blocks to form the lexical tokens it will recognise. Hence, the majority of Lex's functionality is implemented by combining the tokens that are detected by SimpleLex and BasicLex.

"Standard" identifiers, that is, the keywords used in a Test Control Script, are recognised by searching a table of all the built-in identifiers.

The operations of the class are summarised below:

lex(...) - the constructor for initialising all Lex objects at the start of their life.

~lex( ) - the destructor for tidying up at the end of a Lex object's life.

MatchStringOfTokens(...) - From the current point in the input stream attempt to match a sequence of tokens. If required, store the text of the token. (This is useful for 'remembering' numbers and identifiers that were found during a long stream of tokens).

tokenstring(...) - get the text for a built-in token.

next( ) - analyse the next token in the input stream.

skipto(...) - skip through the tokens in the input stream until a matching token is found.

text(...) - get the text for the current token.

readnextline( ) - advance through the input stream until you reach the next line.

reserved( ) - test the current token against the list of reserved identifiers.

line(...) - get the text of the current line in the input stream.

linenum(...) - get the line number of the current position in the input stream.

current( ) - get the last token that was analysed.

A listing of the class is provided in appendix 5.

8.7.2. The Class's Substates

The class's representation is:

```
token tkNow;
```

the remainder of the required representation is inherited from its parent class SimpleLex.

The chosen substate is:

S[1]: the lexical value of the last token analysed by Lex

(1) tkThis equals any of the built in identifier tokens

(2) tkThis equals any of the set of symbols that are recognised by Lex

(3) tkThis equals a number

(4) tkThis equals a string of text surrounded by quotation marks.

(5) tkThis equals a filename surrounded by angle brackets ('<' and '>').

(6) tkthis equals any identifier not including reserved words.

(7) tkThis equals the end of file token.

(8) tkThis equals the token representing unknown tokens in the input stream.

(9) tkThis equals the token representing an error has occurred in the analyser.

8.7.3. The Class's Scenarios

Two scenarios were chosen. They are:

1. an input stream containing many lines of text with the current position of the analyser one token before the end of a line.
2. an input stream with many lines of text with the analyser ready to analyse tokens on the second line.

Both scenarios are used to validate Lex's correct handling of the line-numbers. No extra substates are required.

8.7.4. Test Coverage Results

The test suite generated from the TCS for the lex class contained 122 test cases. The coverage resulting from the execution of these test cases was 52.34 %, that is 67 out of a total of 128 LCSAJs.

The coverage of the individual operations is as follows:

MatchStringOfTokens(...)	= 73.68%	text(...)	= 100.00%
tokenstring(...)	= 66.67%	readnextline( )	= 100.00%
lex(...)	= 100.00%	reserved( )	= 57.89%
~lex( )	= 100.00%	line(...)	= 100.00%
next( )	= 33.80%	linenum(...)	= 100.00%
skipto(...)	= 100.00%	current( )	= 100.00%

The paths that were not executed are as follows:

MatchStringOfTokens(...) :

- The LCSAJ between line 120 and line 126 would be executed if the size if the iSize parameter passed was zero.
- The LCSAJ between line 120 and line 128 is infeasible. bFirstPass is always true on the first iteration of the loop, hence the if statement at line 120 will never evaluate to FALSE on the first pass.
- The LCSAJ between line 126 and line 130 is also infeasible. The path form line 126 onwards would be executed on any iteration of the loop other than the first one. On all iterations other than the first one the if statement at line 128 evaluates to false.

- The LCSAJ between line 133 and line 136 would be executed if we failed to match a symbol on any iteration of the loop other than the first one.
- The LCSAJ between line 143 and line 144 would be executed if the iSize parameter passed to the operation was zero.

tokenstring(...):

- The LCSAJ between line 148 and line 151 would be executed if the token requested is a "built in" identifier (a reserved word).

next( ):

- The LCSAJ between line 178 and line 187 would be executed if the analyser is about encounter the end of the input stream.
- The LCSAJ between line 194 and line 207 would be executed if the analyser encountered a comment in the input stream (comments start with a '#' and continue until the end of the line).
- The LCSAJ between line 209 and line 210 would be executed if the analyser met a token which was a symbol, however it was not the symbol that is used to start a comment (that is, '#').
- The LCSAJ between line 210 and line 213 would be executed if the analyser met an erroneous condition, or token. The condition required for its execution is rare. The path is provided because of defensive programming[25].
- The LCSAJ between line 215 and line 218 would be executed if SimpleLex::next( ) cannot recognise a token. In the current model for SimpleLex, this cannot occur.
- The LCSAJ between line 221 and line 223 is infeasible because the LCSAJ would not terminate at line 223. Line 222 sets bExit to TRUE which causes execution to exit the loop.
- The LCSAJ between line 222 and line 244 would be executed if the analyser met a symbol that was simply an underscore ('_').
- The LCSAJs between lines 222 and 250, 244 and 250, 244 and 256, 221 and 256, and, 222 and 256 would all be executed by different types of identifiers, for example, identifiers starting with and underscore, or identifiers containing a mixture of letters and numbers.

---

25 Defensive programming is a technique where extra code is added to catch all possible error conditions, even though some should never arrise.

- The LCSAJs between lines 256 and 264, 264 and 266, 266 and 268, 268 and 273, 273 and 275, 275 and 280, 280 and 285, 293 and 295, 295 and 297, 297 and 299, 299 and 304, 304 and 309, 309 and 311, 311 and 313, 313 and 315, and, 315 and 317 would each be executed by different single and double character symbols.
- The LCSAJs between lines 317 and 328, 323 and 328, 323 and 334, and, 317 and 334 would be executed if the analyser met strings of different lengths (a string is any sequence of characters surrounded by quotation marks).
- The LCSAJs between lines 317 and 337, 323 and 337, and, 323 and 327 would be executed if the analyser met a string which did not have a terminating quotation mark before the end of the line.
- The LCSAJ between line 336 and line 337 would be executed if the analyser met a valid string (one with a terminator before the end of the line).
- The LCSAJs between lines 337 and 347, 342 and 347, 342 and 353, 337 and 353, 347 and 353, and, 355 and 356 would be executed if the analyser met a valid filename (a strings of tokens bounded by chevrons, '<' and '>').
- The LCSAJs between lines 347 and 356, 337 and 356, and, 342 and 356 would be executed if the analyser met a filename without a terminating chevron.
- The LCSAJ between line 356 and line 358 would be executed if the analyser met a symbol which it did not recognise.
- The LCSAJ between line 365 and line 367 is caused by defensive programming. With the current implementation of SimpleLex, the path is infeasible.

reserved( ) :

- The LCSAJ between line 404 and line 408 would be executed if reserved( ) was called for an identifier which started with a letter which is not in the speeder index (the speeder index is used to jump the search to the start of the appropriate letter).
- The LCSAJs between lines 409 and 410, and, 421 and 422 would be executed if there was an error in the value obtained from the speeder index. This is a form of defensive programming. As the speeder index contains only legal values, this path is effectively infeasible.
- The LCSAJ between line 410 and line 413 would be executed if we failed to find the identifier on the second iteration around the loop. This situation would hence require that the loop was iterated again to continue the search.
- The LCSAJ between line 418 and line 419 would be executed if we were unable to find the identifier in the reserved words list before we reached a word beginning with the next letter.

- The LCSAJ between line 420 and line 422 would be executed if we searched for a reserved word that started with the same letter as the last identifier in the reserved word table, yet we failed to find a valid match.
- The LCSAJ between line 420 and line 423 is infeasible. If execution exited the loop by failing the loop condition, bFound would not equal true.
- The LCSAJ between line 424 and line 425 would be executed if we failed to find the identifier in the reserved word list.

With the infeasible LCSAJs taken into account, the new coverage figure is 67 out of a feasible 124 LCSAJs, that is, 54.03%.

8.7.5. Discussion of Test Results

The class lex achieved a much lower level of LCSAJ coverage than the other classes. This is mainly due to the low code coverage of the next( ) function. This is attributed to both the style in which the operation is written, and the large number of individual values that the attribute tkThis can take. The state-based testing would be over complicated if every value was to be generated. The unexecuted paths of this class can be much more easily validated by other techniques (black, or white-box).

The code could be rewritten to obtain a higher level of coverage from the state-based test suite, however this is not the purpose of testing, hence it will be discussed no further.

8.8. Summary

A suite of state-based test cases have been generated for each of the four case-studies. Varying levels of code coverage have been achieved for different classes. The test suites will be used by the next chapter to evaluate the ability of state-based testing to detect errors that have been seeded into the classes.

The code coverage achieved will be discussed in more detail in the next chapter.

Chapter 9.

Evaluation of State-Based Testing

9.1. Introduction

The previous chapter presented four case studies where state-based testing was used for their validation. The case studies will be used in this chapter to evaluate the viability of state-based testing as a technique for validating object-oriented programs.

The results from the case studies will be analysed for the factors that affected the effectiveness of state-based testing discussed. In addition, two of the classes used for the case studies have been seeded with errors to determine the effectiveness of the technique for the detection of real errors. The results of the error seeding will also be discussed.

Finally, conclusions will be drawn from both the case studies and the error seeding. This will provide details of the situations where state-based testing is effective as a method for the validation of object-oriented programs.

9.2. Effectiveness

Two criteria that can be used to qualitatively measure the effectiveness of a testing technique are: its ability to exercise a class, and its ability to detect errors on the paths exercised. Both of these criteria are evaluated within this chapter. The case studies from the previous chapter will be used to demonstrate the effectiveness of state-based testing for exercising the class under test. The results of each case study will be analysed to determine what factors caused the technique to leave some paths unexecuted. This will be used to conclude the situations where state-based testing is, and is not effective in exercising a class.

The effectiveness of state-based testing for detecting errors will be demonstrated by seeding two of the case studies with meaningful[26] errors and then executing their corresponding test suites. The errors will only be seeded into the executed portions of the class, determined by analysing the results of the coverage achieved when the test suite is applied to the class prior to seeding.

Errors will not be seeded into paths that are not executed by the test suites. As they are unexecuted, any errors seeded into them would be impossible to detect using the state-based test suites.

9.3. Code Coverage

In the last chapter, four case studies were presented, their coverage results will be discussed below. Additionally, the factors that affected the level of coverage achieved will also be discussed.

The first case study to be considered is the linked list of integers. The adjusted[27] coverage achieved by the state-based test suite was 95.96%, or 95 out a feasible 99 LCSAJs. Hence four feasible LCSAJs were not executed (these were described in detail in the previous chapter).

---

26 Meaningful is defined as errors that under certain circumstances will affect the value of data during execution of the classes' operations.

27 After the analysis of the paths that were not executed during the case study, the infeasible ones were removed from the coverage calculation.

The factors that would contribute to the execution of the remaining feasible LCSAJs are:

- The length of the linked list.
- The position of a specific element in the linked list.
- The distance (in number of elements) from the current element to the end of the list.
- When comparing two lists for equality, the linked list that was passed as a parameter is a subset of the current list.

As can be seen, none of the above situations relate directly to values stored in the attributes of the linked list object, although some do relate to the values stored in the dynamic data structure of the object. They refer to different conceptual situations of the linked list object. These situations would be validated either by a black-box test suite which would be concerned with the conceptual states of the linked list rather than the physical ones, or by a white-box test suite that would be concerned with maximising the code coverage of the class.

The second case study was a class for storing strings of text. The state-based test suite achieved 94.79%, or 91 out of a feasible 96 LCSAJs. Hence, 4 LCSAJs were not executed.

The situations that would cause the execution of the currently unexecuted LCSAJs are:

- calling the constructor with a specific value ("" - an empty string)
- search the string forwards from a starting point which is past the end of the string
- search the backwards from a starting point that it before the start of the string.

Calling the constructor with an empty string was not used as a stimuli for the state-based test suite because it is a conceptual state of the parameter, not a physical state. When considering physical states, a pointer to a string is either NULL or not NULL, hence an empty string ("") is no different from a non empty string. This stimuli would be covered by a black-box test suite which would consider the conceptual parameter values.

The final two situations, arising from searching a string when the starting position is past the end of the string in the direction of search, is an error condition within the parameter values, not the values of the class' attributes. Hence a different technique should be used.

The third case study was a simple lexical analyser, the coverage achieved with the state-based test suite was 87.76%, or 43 out of a possible 49 LCSAJs. The situations that would cause the currently unexecuted LCSAJs to be executed are:

- the ordering of a sequence of whitespace characters in the input stream.
- defensive programming
- searching for the current token

The analysis required to determine the precise convoluted order for the whitespace tokens would best be determined during the white-box testing phase.

Although the unexecuted LCSAJs due to defensive programming could be considered infeasible, they are not infeasible due to the logic of the operation. However, they would only be executed under an exceptional circumstance that cannot occur with the current implementation of the class. Defensive programming causes unexecuted paths with other techniques as well (see [56]).

Searching for the current token is a situation that would be covered by a black-box test suite. It was not considered a significant stimuli in comparison to "can the specified token be found", "can the specified token not be found" scenarios.

The final case study was a more complex lexical analyser. The code coverage was considerably lower than the other three case studies, at 54.03% or 67 out of 124 feasible LCSAJs. The situations that would cause the execution of the currently unexecuted LCSAJs are:

- erroneous parameter values
- failing a condition upon any iteration of a loop other than the first
- failure to execute an operation with a specific parameter value
- situations that would be considerably easier to validate using a different technique.
- defensive programming
- errors in the input stream of tokens

All of the above situations are more easily tested using black and white-box techniques. None of them are directly affected by the current state of the object, although some are affected by the current state of parameters.

9.3.1. Conclusions

State-based testing is not a "stand-alone" technique for providing complete validation of object-oriented programs, hence it is unlikely to cause 100% of the feasible LCSAJs of a class to be executed. However it is effective in exercising those parts of operations that interact with the state of an object. The high level of coverage achieved with 3 out of the 4 case studies implies that the coverage of classes with a high degree of dependency between the operations and the state will be high. The remainder of the coverage for all four case studies can be achieved using black and white-box techniques[28].

The low level of coverage achieved in the fourth case study is easily attributed to the fact that the execution of the majority of LCSAJs is not affected by the current state of the object, it is affected by the current contents of the input stream. State-based test cases could have been generated, however this would have complicated the state-based testing process considerably. The situations are far easier to validate using a black-box technique.

Defensive programming is one of the most significant factors in causing LCSAJs to remain unexecuted. This affects not only state-based testing, but also other testing techniques. [56]

9.4. Error Seeding

In the previous section we have discussed the effectiveness of state-based testing in causing code coverage of classes under test. However, code coverage is not a guarantee that the errors within the code have been detected, even on paths that were executed by the test cases. Hence, there is a requirement for a further evaluation of the technique to determine its effectiveness for detecting errors on paths that have been exercised. This evaluation will take the form of error seeding, where errors are purposefully introduced into the code to determine if the test suite is adequate enough to detect them. The following sections will describe the errors seeding.

---

[28] 100% coverage of the feasible LCSAJs has been achieved using a combination of all three types of techniques.

9.4.1. The Seeding Method

Two of the classes from the case studies have been seeded with errors. The classes were seeded with one error at a time, then the whole state-based test suite was executed and the results gathered. All the errors that were seeded were live, that is, they affected the value of data used during the execution of the operations, they were not pragmatic changes to the code such as the renaming of variables. Also, the errors were syntactically and semantically correct, hence they would compile and allow the test case to execute.

The types of errors seeded can be classified as follows:

Missing actions - These are seeded by simply removing statements from the source code. The statements removed should not directly affect the control-flow of the operation, although they may influence it indirectly by having changed the value of data used within the operation.

Missing paths - These involve the alteration or deletion of statements that directly affect the control-flow during execution of the operation.

Missing data - These errors are seeded by removing data that is required by the operation; an example would be the removing of data from a lookup table used by the operation.

Additional actions - These errors are seeded by introducing statements that do not directly affect the control-flow of the operation, although they may alter the value of data used within the operation, which in turn may indirectly affect the control-flow of the operation.

Additional paths - Additional path errors involve the insertion of statements that directly affect the control-flow of the operation. Examples of these would include conditional statements such as 'if' and 'while' or control-flow statements such as 'goto' and 'return'.

Additional data - These errors are created by the addition of extra data to data that is required by the operation. An example would be the addition of data to a lookup table used by an operation.

Altered actions - These are created by altering an action performed by a single statement within an operation. An example would be the changing of an addition operator to a subtraction operator.

Altered data - Similarly, altered data errors are created by changing the value of data used by the operation. For example, an increment of one could be changed to an increment of 3. These errors are very similar to altered actions, hence on occasion they are indistinguishable.

This classification described above is an extension to one presented by Howden in [68, 69]. The errors for the classes were seeded in two halves. The first half were seeded by the author, the second half were seeded by a member of the Visual C++ team at Microsoft Corp., Redmond, USA. The author had no knowledge of the errors seeded at Redmond. Only one error was seeded into the class at any one time, removing the potential for errors co-operating together to avoid detection.

When choosing the actual errors to be seeded into the code, an attempt was made to increase the realism of the evaluation by choosing errors and situations that were similar to actual errors that have been found during the validation of other programs and classes.

9.4.2. The Seeding

The first class to be seeded was the linked list of integers, 20 errors in total were seeded into the various operations of the class. Only one error was present in the class for each test run, and all operations of the class were seeded with at least one error during the evaluation. However, the class was not seeded with errors of the 'missing data' or 'additional data' type, as the class does not use any static data such as lookup tables during its execution.

The state-based test suite failed to detect 3 out of the 20 errors (numbers 11,14 and 18), hence the technique was 85% effective in detecting errors present on the paths that can be executed by this test suite. A discussion of the above results will be provided in the next section.

The second class to be seeded with errors is the lexical analysis class - lex. Again, 20 errors were seeded into the paths executed by the state-based test suite. As with the linked list class, only one error was present in the class during any test run.

The state-based test suite detected all 20 errors that were seeded into the class. Hence, for this class, the test suite was 100% effective.

9.4.3. Error Seeding Results Analysis

The first testrun failed to detect three of the 20 errors that were seeded into the class. The errors that were not detected were as follows:

Error 11:

```
[1]   CDT_Slist::CDT_Slist(const CDT_Slist & Copy)
[2]   {
[3]      pTop = NULL;
[4]      //   pCur = &pTop; // ERROR 11
[5]      pCur =NULL;
[6]      // pass the buck to the assignment operator
[7]      *this = Copy;
[8]   }
```

This error is on line [4] of the copy-constructor[29] and is a missing action which should have set substate 3 to value 1. The reason the error remained undetected is because it is not actually needed for the correct operation of the class. Line [7] calls the assignment operator to perform the majority of the functionality for creating an object as a copy of another one. The first action of the assignment operator is to delete all the nodes of the current linked list and to reinitialise the list as an empty one. This reinitialisation also sets substate 3 to value 1, therefore making the missing statement (the error) superfluous.

error 14:

```
[1]          if (pLast == NULL)
[2]          {
[3]             pTop = pTmp->plNext;
[4]   //                delete pTmp; // ERROR 14
[5]             pCur = &pTop;
[6]          }
```

Error 14 on line [4] is also a missing action. The above code is taken from the remove(...) operation.

---

[29] A copy-constructor in C++ is a constructor that creates an instance of a class by copying the information from another object that already exists

The error was not detected because the missing action does not have any effect on the substates of an object. The action deallocates a node in the linked list and returns the memory to the system heap. This error could only be detected by test cases that monitor the current level of memory allocation within the list, this type of test case would be easier to generate using techniques other than state-based testing.

Error 18:

```
[1]   // ERROR 18 (the next line used to include iLoop = 1)
[2]   for (int iLoop = 3; iLoop <= Compare.size(); iLoop ++)
[3]   {
[4]      if (lCopy.remove(pTmp->item()) == FALSE)
[5]         return FALSE;
[6]      pTmp = pTmp->plNext;
[7]   }
[8]   return TRUE;
```

As can be seen above, the error is of an altered action/data type, where the iLoop variable is initialised to the value 3, instead of the value 1. Again, neither this action, nor the code following it have any effect on the substate values of the current object, hence the missing action cannot be detected by the state-based technique.

9.4.4. Conclusions

Both state-based testing suites were very effective in detecting the errors that were seeded into the paths covered by the test suites. The test suite for the first class detected 85% of the errors seeded into the class, the test suite for the second detected 100% of the errors seeded. The majority of these paths either use or set the control-information within the state of the object, hence errors in these paths have a greater chance of detection. Some errors that affect the data-storage information within the state can also be detected, however because the main emphasis of a state-based test suite is the control-information.

9.5. Summary

In the previous chapter (chapter 8.), the effectiveness of state-based testing for causing code coverage within the operations of some classes was demonstrated. These results were used within this chapter to determine some of the factors affecting the code coverage achieved with a state-based test suite. This confirmed that the emphasis of state-based testing is indeed the control-information stored within an object's state. An object with a higher dependence on the control-information stored within itself will achieve a greater coverage with a state-based test suite than a class with a low dependence upon its control-information.

State-based testing is effective in discovering errors that are seeded on the paths that can be reached by a state-based test suite. The errors that affect the state of the class under test will be detected more readily than those errors that have no effect on the state of the class under test.

Chapter 10.

Conclusions

10.1. Thesis Summary

This thesis has investigated current testing techniques for object-oriented programs to discover some of the problems involved. Of these techniques, very few have been designed specifically for use with object-oriented programs, the majority are adaptations of more traditional approaches.

The technique described in this thesis has addressed a major problem found with current techniques. State changes are an essential part of object-oriented programming. State-based testing emphasises the state changes that take place within an object during its lifetime. It has been developed for use in conjunction with other testing techniques and is therefore complementary to many traditional testing techniques adapted for object-oriented programs.

Within a class the values that can be stored by the attributes can be divided into two groups, control-information and data-storage values. State-based testing concentrates upon the control-information, modelling its transitions as a finite state-automaton. The automaton describes the state changes that should occur when each operation is invoked with particular parameter values. These state changes are validated to ensure that at no point in an object's life-cycle can the invocation of an operation result in an undefined or invalid state. These state-changes are validated directly by interacting with the attributes, thus reducing the need other operations which have not yet been validated.

The values used during the validation are obtained from an analysis of both the specification and the design of the class. These values are actual values used by the attributes of the class rather than conceptual ones. The use of physical values eases their analysis and their validation as there is less room for discretion. Moreover, conceptual values are difficult to analyse and very difficult to validate without using other operations from the class' interface.

The basic state-based technique is enhanced with the incorporation of scenarios. A scenario is a particular situation that is specific to the model upon which the class is based. A test case using a scenario as a starting state involves the creation of a highly specific situation, thus increasing the testers control over the situations that are validated. The finishing states are validated in the same manner as the more basic test cases. Scenarios may be used to represent conceptual situations that may not map easily onto the values used for the basic state-based test cases.

State-based testing can be applied to object-oriented programs during both the unit (class) and the integration phases of testing. At the unit level, it should be applied in conjunction with black and white-box techniques to provide a more complete validation of programs. Classes defined with inheritance are also catered for by an adaptation to the incremental testing algorithm of Harrold et al. [55]. Only those inherited operations affected by any changes in the use of inherited attributes are retested. The incorporation of state-based testing into this algorithm aids in the detection of these changes in use.

A prototype suite of tools has been developed facilitating an evaluation of the technique. The tools suite consists of three tools, MKTC the main test case generator, MKMFTC a utility to aid their automatic compilation, and TESTRUN a utility to automate the execution and results gathering from large numbers of test cases.

Four different classes have been used for the evaluation, a state-based test suite was generated for each. When the test suite was executed, the LCSAJ code coverage of the class was measured. The code coverage was used as a metric for measuring the amount of the class exercised by the test suite, and thus the technique. These results suggest the technique is effective in achieving coverage of classes, the extent of the coverage is dependent upon the degree of dependency between the operations and the control-information. The average coverage for the classes used in the evaluation was 83.4% of the feasible LCSAJs.

Two of the case studies were then seeded with errors to determine the effectiveness of the technique for detecting errors present on paths that are executed by the test suite. over 95% of these errors were

discovered. Those that were not discovered either did not affect the functionality of the class, or had no effect upon the control-information stored and used by the class.

10.2. Critical Assessment

The requirements for a new testing technique for validating object-oriented programs (as outlined in section 4.5) are addressed below. They were (in summary):

1. the technique must address fundamental conecpts of object-oriented programming,
2. it must consider the state of an object during testing,
3. it must be possible to test operations in isolation from each other,
4. it must be methodical to apply,
5. it must have an easily determined end point when no more test cases can be generated,
6. it must be effective in locating errors with a near optimal suite of test cases,
7. it should use physical values that are used within the objects rather than conceptual ones,
8. it must be possible to automate the application of some if not all of the technique.

State-based testing is designed to validate the state changes that can occur within the state of an object (requirement 2). More specifically, the state changes occurring within the control-information of the class. Those parts of the class that are not directly concerned with the control-information will not be exercised directly. This is not to say that they would not be exercised at all, however, errors in these parts of a class are less likely to be detected by a state-based test suite. The greater the level of control information within a class, the greater the code coverage achieved, and the more effective validation is using state-based testing.

The technique has been designed to validate a specific aspect of object-oriented programming (requirement 1). It uses physical values to model the state-changes within the class (requirement 7), easing its application to classes when compared with black-box techniques such as ASTOOT that require the use of conceptual states. The application of equivalence partitioning to the range of physical values dramatically reduces the number of values that need be manipulated during test cases. This is based upon the assumption that all values within a partition are treated in a similar manner by the code under test. Although this assumption reduces the effectiveness of the testing technique in theoretical terms, it dramatically reduces the number of test cases needed and therefore increases the number of test cases that can be executed given limited testing resources, and is therefore more considerably more applicable in a practical situation.

The evaluation demonstrated that the technique is effective in uncovering errors that exist on paths executed by the test suite (requirement 6). Of the errors seeded into two classes, all except three were discovered. Although the study was a relatively small sample of classes, the number of errors seeded was high in relation to the number of lines of code. This implies that the technique is effective and hence satisfies its original criteria - that of providing effective validation for the state changes that occur within the state of an object.

When compared with other techniques such as the method of Hoffman and Strooper (see [63, 131]), or Frankl and Doong's ASTOOT suite of tools (see [42, 35]), state-based testing produces a factor decrease in the number of test cases generated thus the number of test cases are more optimal than with ASTOOT (requirement 6). However, state-based testing does not guarantee 100% code coverage and therefore cannot be used alone, hence additional test cases will be required. Nevertheless, the other techniques imply they provide complete validation although they are both black-box and therefore cannot even guarantee 100% statement coverage.

State-based testing cannot provide complete validation of object-oriented programs alone, it must be used in conjunction with other complementary techniques. Black-box techniques such as equivalence partitioning, or cause-effect graphing, along with white-box techniques such as data-flow testing, or branch coverage can be used to enhance the code coverage already achieved by state-based testing to provide adequate testing of object-oriented programs.

From the results discussed in the evaluation chapter, it is apparent that some classes will be tested far more effectively if they are tested using techniques other than state-based testing. The more complex of the two lexical analysers used for the evaluation would be validated more effectively with other techniques such as the black-box finite state automata method of Beizer [9], or branch testing, than with state-based testing.

No technique alone can expect to provide complete validation of programs and hence this is not a necessarily a drawback. Very few testing techniques can be easily and effectively applied to all programs, hence the benefit of any technique arises in its application to those classes that it is best suited to; for state-based testing, those that manipulate large amounts of control-information are ideal.

For the technique to be applied effectively, it relies heavily on the tester's expertise for analysing the values required, and for determining the state changes that should occur within the state of the object. The analysis of the values cannot be made any more formal because the information that is used to derive the values is often informal by nature. Formal and informal specifications may provide information about the conceptual uses of a class, however physical values must be obtained from the

class' design. Because this procedure is informal, slightly differing results will be obtained by different testers. Although with experience, the level of testing achievable with the state-based technique will improve. State-based testing is not alone, the majority of testing techniques rely upon the expertise of the tester in applying them effectively.

The technique does not include any facilities for testing parallel object-oriented programs. Parallelism involves additional problems such as deadlock and race conditions, hence any new technique must encompass considerations from both the object-oriented and parallel programming research areas. In theoretical terms, it would require an extension of the technique from Σ^+ to Σ^*. This extension implies that objects can change state without any external stimuli (particularly under the control of the tester); without this controlled change state-based testing cannot be applied. This restriction will unacceptably reduce the situations that it will be possible to exercise the software through.

If it is possible to model each object within the parallel system as a state with a sequence of synchronous events (that is, events that occur under a defined control pattern), then it may be possible to apply state-based testing in its simplest form. Synchronisation would have to be added to objects for this type of validation to be possible. The synchronisation would allow a test case to know the object's state at the start of the test case by synchronising with it. If the object reacts to external events such as operation calls, or hardware events, these can be monitored and administered as required by each test case. A further synchronisation point would allow a result to be obtained. Testing in this manner would exercise particular code paths between the starting and finishing synchronisation points, although it would not provide "real-world" situations where the system does not run only between two synchronisation points. It also doesn't cover cases where an operation provided by an object is invoked multiple times in parallel.

State-based testing provides no special facilities for handling polymorphism or dynamic binding. Dynamic binding and polymorphism are concerned with selecting the correct operation depending upon the type of the object that the operation is being invoked upon. This is of considerable concern during the integration testing of objects. However, state-based testing is not directly applicable to this type of problem. Errors in these areas of a program will affect the invocation of the correct operation, whereas state-based testing is concerned with the transitions that occur as a result of a specific operation.

The generation of test cases by hand for any practical application is infeasible, hence for the technique to be easily applicable, it requires extensive automation in the form of support tools. The current suite of tools provides only a very basic level of support for the technique, as they were primarily written to facilitate the technique's evaluation. They provide the facility for automatically generating test cases

from a script that describes the state transitions to be validated (requirement 8). These test cases are automatically compiled and executed. However the tools require a considerable amount of disk space and processing time to be able to complete either of these tasks.

The tools are unable to generate an infinite number of unique test cases because of the limited number of permutations that the state of an object can undergo. When performing state-based testing, the user must apply equivalence partitioning to the values used within the attributes of the object under test. The partitioning reduces the potentially infinite range of values that must be manipulated down to a more manageable range. This not only makes the testing more practically applicable, but also restricts the number of test cases that can be generated to a set of finite size (requirement 5).

In summary, state-based testing has satisfied all of the requirements outlined in section 4.5.

10.3. Future Directions

As mentioned above, currently the technique does not attempt to validate any polymorphic behaviour within programs. Polymorphism does not easily fit into the model upon which state-based testing is based. An alternative would be to develop an additional technique to validate this and other aspects of object-oriented programming that are not adequately covered by current testing techniques.

As discussed in the previous section, some classes are more easily and effectively validated with other techniques. An extensive evaluation is required to provide an accurate set of guidelines that can easily determine which classes should be tested by state-based testing and which ones should not. This evaluation would require careful analysis of many aspects of classes each class' functionality and implementation.

The majority of the technique is simple and methodical to apply, however there is an exception: the analysis of a class' substates and substate-values is currently very dependent upon both the level of design information available and the testers ability to analyse it for the required values. Guidelines were provided in Chapter 5 for the analysis of the data, however these guidelines are relatively simplistic and a tester with experience will be able to see obvious cases and situations that are not normally found using the guidelines. Therefore, a considerable work is required to extend and formalise the guidelines for choosing substate-values.

One potential improvement to the guidelines would be to link the testing control script to various automated CASE/design tools that are commercially available. This would require care consideration

of the design methods to determine the most suitable for use in cooperation with state-based testing. It would be advantageous to choose a technique that allows access to the data from outside of the tool suite, therefore a filter could be produced for partially if not completely generating the test control script for driving the test case generation.

The analysis for substate-values should be performed on the design (preferably the low level design), rather than the code itself. If the low level design is not available, then values can potentially be obtained from careful analysis of the code. The values obtained from the code will provide a rough guideline for the choice of substate-values. The values obtained in this manner cannot be guaranteed for accuracy, as the code is currently under test. It is always preferable to analyse the design, as the code is then validated against the code (and ultimately the requirements).

The technique has only been discussed in connection with small programs and classes within this thesis. Research is required to determine the scalability of the technique for large complex systems. A number of problems are currently evident when viewing the scalability: the creation, manipulation and validation of large numbers of objects that are required for testing large systems, the difficulty in analysing the substate-values to be used (see discussion above), the complexity of determining the substate combinations required for test cases, the time required to specify the test cases, and the storage and processing requirements needed for the execution of the test cases.

As testing proceeds up a project's object hierarchy, the number of objects that must be created, manipulated and validated increases with an exponential growth. For large and complex systems the number of objects becomes a considerable problem that must be addressed by all testing techniques and state-based testing is no exception. Any reductions that can be made to the number of test cases generated, and the resources needed to generate and execute them will enable more resources to be devoted to those critical sections of a project. The problems associated with testing in a large system must be addressed by any future research on this technique.

The objects used in the evaluation were relatively small (less than 15 operations), as the number of attributes and operations grows, so too does the complexity of the control-information interactions that occur. As yet it is unknown whether these interactions will become too complex for practical testing with state-based testing. This must be determined by any further research.

The time required to specify the test cases to an automated tool could also be reduced by cooperation with an automated design tool. Design tools can be used to describe the models upon which classes are based. These models can be used to aid in the generation of test control scripts. Research is required

determined the requirements state-based testing has upon the information provided by any such design tools. Also, research is required to determine whether any such design tools currently exist.

Changes to the tools suite that are required include:

- Disk usage - as mentioned in the previous section, each test case includes the same set of classes and hence there is an extensive replication of executable code. More efficient methods for implementing the test cases are required without sacrificing the ability to run individual test cases easily.
- Integration testing - currently there is no support for any form of integration testing. This would involve the tool reading multiple test control scripts to determine the substate for all object involves. Generators would already exist for creating objects in specific states. These could be used for the generation of parameters that are passed to other objects under test.
- User friendliness - in keeping with current trends towards more user friendly graphical user interface based tools, the human - tool interface could be much improved. The test control scripts are very concise at expressing test cases, however, they are slow to generate and error prone. By using an interface that guides the tester through the process, numerous errors could be avoided.
- Error reporting - in keeping with the UNIX nature of the tools, the error reporting of the tools is terse and sometimes unhelpful.
- Inheritance - currently the tester has to cut and paste portions of the test control script from the parent classes into the test control script for the child class. A large portion of this process could be automated, improving the support for derived classes and inheritance hierarchies.

Extensive evaluation is required to determine the techniques that gain the maximum benefit when combined with state-based testing. In addition, research is needed to reduce the overlap between the test cases suites generated by these techniques and state-based testing.

Chapter 11.

References

[1] Agha, G., An Overview of Actor Languages, *SIGPLAN Notices*, vol. 21, no. 10, pp. 58 - 67, October 1986

[2] Aksit, M., Dukstra, j. W. and Tripathi, A., Atomic Delegation: Object-Oriented Transactions, *IEEE Software*, pp. 84 - 92, March 1991

[3] Al-Haddad, H. M., George, K. M. and Samadzadeh, M. H., Approaches to reusability in C++ and Eiffel, *Journal of Object-Oriented Programming*, pp. 34 - 45, September 1991

[4] Archie, K. C. and McLear III, R. E., Environments for Testing Software Systems, *AT&T Technical Journal*, pp. 65 - 75, March/April 1990

[5] Backus, J., Can Programming Be Liberated From the Von neumann Style ? A Functional Style and its Algebra of Programs, *Communications of the ACM*, vol. 21, no. 8, pp. 613 - 641, August 1978

[6] Balin, S. C., An Object-Oriented Requirements Specification Method, *Communications of the ACM*, vol. 32, no. 5, pp. 608 - 623, May 1989

[7] Balzer, R. Goldman, N. and Wile, D., Informality in Program Specifications, *IEEE Transactions on Software Engineering*, vol. SE-4, no. 2, pp. 94 - 103, March 1978

[8] Bavel, Z., *Introduction to the Theory of Automata*, Reston Publishing Company (Prentice-Hall), 1983

[9] Beizer, B., *Software Testing Techniques*, 2nd Edition Ed., Van Nostrand Rienhold, New York, USA, 1990

[10] Berard, E., Object Oriented Testing, *published in USENET group comp.object*, November 1990

[11] Berard, E., Specifying Test Cases for Object-Oriented Systems, *published in USENET group comp.object*, October 1991

[12] Berard, E. V., *Tutorial 3.0: Testing Object-Oriented Software*, Notes from tutorial presented at OOPSLA 92, Vancouver, Canada

[13] Berard, E., *Essays on Object-Oriented Software Engineering*, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1993

[14] Bertolino, A., An Overview of Automated Software Testing, *Journal of Systems and Software*, vol. 15, pp. 133 - 138, 1991

[15] Birtwistle, G. M., Dahl, O. J., Myhrhaug, B. and Nygaard, K., *Simula Begin*, Studentlitteratur, Lund, Sweden, 1979

[16] Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L., Distribution and Abstract Types in Emerald, *IEEE Transactions on Sfotware Engineering*, vol. SE-13, no. 1, pp. 65 - 76, January 1987

[17] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F., CommonLoops Merging Lisp and Object-Oriented Programming, in *Proceedings of the*

Object-Oriented Programming Systems, Languages and Applications Conference, pp. 17 - 29, SIGPLAN Notices, ACM Inc., New York, USA, 1986

[18] Boehm, B., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981

[19] Booch, G, *Software Engineering with Ada*, 2nd Ed., Benjamin Cummings, 1987

[20] Booch, G., *Object Oriented Design With Applications*, Benjamin Cummings, 1991

[21] Borning, A. H., Classes versus Prototypes in Object-Oriented Languages, in *Proceedings of the Fall Joint Computer Conference*, pp. 36 - 40, IEEE, 1986

[22] Budd,T.A. and Lipton, R. J., Mutation Analysis of Decision Table Programs, in *Proceedings of the Conference on Information Science snd Systems*, 1978

[23] Budd T. A., Lipton, R. J., Sayward, F. G. and DeMillo R. A., The Design of a Prototype Mutation System for Program Testing, in *Proceedings of the National Computer Conference*

[24] Cardelli, L., A Semantics of Multiple Inheritance, in *Semantics of Data Types*, by Kahn, G. et al., pp. 51 - 67, Springer Verlag, 1984

[25] Cato, J. and Warren, J., Object-Orientation and Information Management, in *Proceedings of the Conference on Software Tools*, pp. 97 - 103, Online Publications, Pinner - Adv. computing series number 8, 1987

[26] Chambers, C., Ungar, D., Chang, B. and Hölzle, U., Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF, *LISP and symbolic computation: an international journal*, vol. 4, no. 3, pp. 21 - 36, 1991

[27] Cheatham T. J. and Mellinger L., Testing Object-Oriented Systems, in *Proceedings of the 18th ACM Annual Computer Science Conference*, pp. 161-165, ACM Inc., New York, USA, 1990

[28] Chow, T. S., Testing Software Design modelling by Finite-State Machines, *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178 - 187, May 1978

[29] Cohen, A. T., Data Abstraction, Data Encapsulation and OOP, *SIGPLAN Notices*, vol. 19, no. 1, pp. 31 - 35, January 1984

[30] Cook, S., Languages, and Object Oriented Programming, *Software Engineering Journal*, pp. 73 - 80, March 1986

[31] Coward, P. D., A review of Software Testing, *Information and Software Technology*, vol. 30, no. 3, pp. 189 - 198, April 1988

[32] Cox, B. J., *Object-Oriented Programming An Evolutionary Approach*, Addision-Wesley, Reading, Massachusetts, USA, 1986

[33] Dahl, O. -J. and Nygaard, K., Simula: An Algol-Based Simulation Language, *Communications of the ACM*, vol. 9, no. 9, pp. 671 - 678, September 1966

[34] Darringer, J. A. and King, J. C., Applications of Symbolic Execution to Program Testing, *Computer*, pp. 51 - 60, April 1978

[35] Doong. R. K. and Frankl, P., Case Studies in Testing Object-Oriented Programs, in *Proceedings of the 4th Testing, Analysis and Verification Symposium*, pp. 165 - 177, ACM Inc., New York, New York, 1991

[36] Dorman, M., Unit Testing C++ Objects, in *Proceedings of EUROSTAR*, pp. 21A/1 - 21A/31, 1993

[37] Duran, J. W., A Report on Random Testing, in *Proceedings of the 5th International Conference on Software Engineering*, pp. 179 - 183, IEEE, 1981

[38] Ellis, M. A. and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, Reading,Massachusetts, 1990

[39] Embley, D. W. and Woodfield, S. N., Cohesion and Coupling for Abstract Data Types, in *Proceedings of the IEEE Phoenix Conference on Computers and Communications*, pp. 229 - 234, IEEE, 1987

[40] Fagan, M., Design and Code Inspection to Reduce Errors in Program Development, *IBM Systems Journal*, vol. 15, no. 3, 1976

[41] Fiedler, S. P., Object-Oriented Unit Testing, *Hewlett-Packard Journal*, pp. 69 - 74, April 1989

[42] Frankl, P. G. and Doong, R., Tools for Testing Object-Oriented Programs, in *Proceedings of the 8th Pacific NorthWest Conference on Software Quality*, pp. 309 - 324, 1990

[43] Gannon, J. D., Data Types and Programming Reliability : Some Preliminary Evidence, in *Proceedings of the 1st Symposium on Computer Software Engineering*, pp. 367 - 376, IEEE, 1976

[44] Gannon, J., McMullin, P. and Hamlet, R., Data-Abstraction Implementation, Specification, and Testing, *Transactions on Programming Languages and Systems*, vol. 3, no. 3, pp. 211 - 223, July 1981

[45] Gehani, N., Specifications: Formal and Informal - A Case Study, *Software - Practice and Experience*, vol. 12, pp. 433 - 444, 1982

[46] Goldberg, A. and Robson, D., *Smalltalk 80: The language and its Implementation*, Addison Wesley, Reading, Massachusetts, USA, 1983

[47] Goodenough, J. B. and Gerhart, S. L., Towards a Theory of Test Data Selection, *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 156 - 175, June 1975

[48] Graham, J.A. Drakeford, A. and Turner C.D., The Verification, Validation and Testing of Object-Oriented Systems, *BT Technology Journal*, vol. 11, no. 3, pp. 79 - 88, July 1993

[49] Grogono, P. and Bennett, A., Polymorphism and Type Checking in Object-Oriented Languages, *SIGPLAN Notices*, vol. 24, no. 11, pp. 109 - 115, November 1988

[50] Guttag, J. V., Horowitz, E. and Musser, D. R., Abstract Data Types and Software Validation, *Communcations of the ACM*, vol. 21, no. 12, pp. 1048 - 1064, December 1978

[51] Halbert, D. C. and O'Brien, P. D., Using Types and Inheritance in Object oriented Programming, *IEEE Software*, pp. 71 - 79, Spetember 1987

[52] Haley, A. and Zweben, S., Development and Application of a White-Box Approach to Integration Testing, *Journal of Systems and Software*, pp. 305 - 315, 1984

[53] Hamlet, D. and Taylor, R., Partition Testing Does Not Inspire Confidence, *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402 - 1411, December 1990

[54] Harrold, M. J. and Soffa, M. L., Selecting and Using Data for Integration Testing, *IEEE Software*, pp. 58 - 65, March 1991

[55] Harrold M. J., McGregor, J. D. and Fitzpatrick K. J., Incremental Testing of Object-Oriented Class Structure, in *14th International Conference on Software Engineering*, ACM, 1992

[56] Hedley, D. and Hennell, M. A., The Cause and Effects of Infeasible Paths in Computer Programs, in *Proceedings of the 7th International Conference on Software Engineering*, pp. 266 - 277, IEEE, 1984

[57] Henderson-Sellers, B and Edwards, J. M., The Object Oriented Systems Life Cycle, *Communications of the ACM*, vol. 33, no. 9, pp. 142 - 159, September 1990

[58] Hendler, J. A. and Wegner, P., Viewing Object-Oriented Programming as an Enhancement of Data-Abstraction, in *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, pp. 117 - 125, 1986

[59] Hennell, M. A., Hedley, D. and Woodward, M. R., The Implications of a Hierarchy of Coverage Measures on Program Testing, *Tech. Rep.* , Dept. of Computer Science, Liverpool University, England, 1981

[60] Herington, D., E., Nichols, P., A. and Lipp, R., D., Software Verification Using Branch Analysis, *Hewlett-Packard Journal*, vol. 38, no. 6, pp. 13 - 23, June 1987

[61] Hoare, C. A. R., Proof of Correctness of Data Representations, *Acta Informatica*, pp. 271-281, 1972

[62] Hoffman, D., A CASE Study in Module Testing, in *Proceedings of the Conference on Software Maintenance*, pp. 100 - 105, IEEE, 1989

[63] Hoffman, D. and Strooper, P., Graph-Based Module Testing, in *Proceedings of the 16th Australian Computer Science Conference*, pp. 479 - 487, Australian Computer Science Communications, Queensland University of Technology, Australia

[64] Hoffman, D. and Strooper, P., Graph-based Class Testing, in *Proceedings of the 7th Australian Sofware Engineering Conference (ASWEC)*, 1993

[65] Hoffman, D. and Brealey, C., Module Test Case Generation, in *Proceedings of the 3rd Symposium on Testing, Analysis and Verification*, pp. 97, 102, Software Engineering Notes, Volume 14, Number 8, ACM SIGSOFT, 1989

[66] Howden, W. E., Reliability of the Path Analysis Testing Strategy, *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 208 - 215, September 1976

[67] Howden, W. E., Symbolic Testing and the DISSECT Symbolic Evaluation System, *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 266 - 278, July 1977

[68] Howden, W. E., Functional Program Testing, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 162 - 169, March 1980

[69] Howden, W. E., The Theory and Practice of Functional Testing, *IEEE Software*, pp. 6 - 17, September 1985

[70] Howden, W. E., *Functional Program Testing and Analysis*, McGraw Hill, New York, USA, 1987

[71] IEEE Standard 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990

[72] Ingalls, D. H. H., Design Prociples Behind Smalltalk, *BYTE*, vol. 6, no. 8, August 1981

[73] Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G., *Object-Oriented Software Engineering : A Use Case Driven Approach*, ACM Press (Addison-Wesley), 1992

[74] Jalote, P., Specification and Testing of Abstract Data Types, in *Proceedings of COMPSAC*, pp. 508 - 511, IEEE Computer Society Press, 1983

[75] Jalote, P. and Caballero, M. F., Automated Test Case Generation for Data Abstraction, in *Proceedings of COMPSAC*, pp. 205 - 210, IEEE Computer Society Press, 1988

[76] Jalote, P., Testing the completeness of Specifications, *IEEE Transcations on Software Engineering*, vol. 15, no. 5, pp. 526 - 531, May 1989

[77] Johnson, R. E. and Zweig, J. M., Delegation in C++, *Journal of Object-Oriented Programming*, pp. 31 - 34, November/December 1991

[78] Jorgensen, P. C., MM-Paths : A White-Box Approach to Software Integration Testing, in *Proceedings of the 3rd Annual IEEE Phoenix Conference on Computers and Communications*, pp. 181 - 185, IEEE, 1984

[79] Joyce, D., An Identification and Investigation of Software Design Guidelines Using Encapsulation Units, *The Journal of Systems and Software*, pp. 287 - 295, 1987

[80] Kay, A. C., The Early History of Smalltalk, in *Proceedings of the History of Programming Languages Conference (HOPL-II)*, pp. 69 - 90, ACM SIGPLAN Notices Volume 28, Number 3, 1993

[81] Kent, W., A Rigorous Modle of Object Reference, Identity and Existence, *Journal of Object-Oriented Programming*, pp. 28 - 34, June 1991

[82] Khoshafian, S. N. and Copeland, G. P., Object Identity, in *Proceedings of the Object Oriented Languages, Systems and Applications Conference*, pp. 406 - 416, ACM SIGPLAN Notices , 1986

[83] Kilian, M., Trellis: Turning Designs into Programs, *Communications of the ACM*, vol. 33, no. 9, pp. 65 - 67, September 1990

[84] Korson, T. and McGregor, J. D., Understanding Object-Oriented :- A Unifying Paradigm, *Communications of the ACM*, vol. 33, no. 9, pp. 40 - 60, Spetember 1990

[85] LaLonde, W. and Pugh, J., Subclassing doesn't equal subtyping doesn't equal is-a, *Journal of Object-Oriented Porgramming*, pp. 57 - 62, January 1991

[86] Lang, K. J. and Pearlmutter, B. A., Oaklisp: an Object-Oriented Scheme with First Class Types, in *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 30 - 37, ACM SIGPLAN Notices, 1986

[87] Laski, J., On Data-flow Guided Program Testing, *ACM SIGPLAN Notices*, vol. 17, no. 9, pp. 62 - 71, September 1982

[88] Laski, J. W. and Korel, B., A Dataflow-Oriented Program Testing Strategy, *IEEE Transactions on Software Engineering*, pp. 1483 - 1498, May 1983

[89] Leavens, G. T., Modular Specification and Verification of Object-Oriented Programs, *IEEE Software*, pp. 72 - 80, July 1991

[90] Lee, S. and Carver, D. L., Object-Oriented Analysis and Specification: a Knowledge Base Approach, *Journal of Object-Oriented Programming*, pp. 35 - 43, January 1991

[91] LeJacq, J. P., Function Poreconditions in Object-Oriented Software, *ACM SGIPLAN Notices*, vol. 26, no. 10, pp. 13 - 18, October 1991

[92] Leung, H. K. N. and White, L., J., A study of Integration Testing and Regression Testing, *Tech. Rep. TR 89-21*, Department of Computer Science, University of Alberta, Canada, 1989

[93] Lieberherr, K. J. and Holland, I. M., Assuring Good Style for Object-Oriented Programs, *IEEE Software* , pp. 38 - 48, September 1989

[94] Lieberherr, K. J. Bergstein, P. and Silva-Lepe, I., From Objects to Classes: Algorithms for Optimal Object-Oriented Design, *Software Engineering Journal*, pp. 205 - 228, July 1991

[95] MacLennan, B. J., Values and Objects in Programming Languages, *ACM SIGPLAN Notices*, vol. 17, no. 12, pp. 70 - 79, December 1982

[96] Madsen, O. L. and Møller-Pedersen, B., What Object-Oriented Programming May Be - and What It Does Not Have To Be, in *Proceedings of the European Conference on Object-Oriented Porgramming (ECOOP)*, pp. 1 - 20, Springer-Verlag, 1988

[97] McCabe, T. J., *Structured Testing*, IEEE Computer Press, 1983

[98] McCabe, T., *Testing: Past, Present and Future Including C++*, Presentation at BT Laboratories, Martlesham Heath, Ipswich, March 1993

[99] Meyer, B., Reusability: The Case for Object Oriented Design, *IEEE Software*, vol. 4, no. 2, pp. 50 - 64, March 1987

[100] Meyer, B., *Object Oriented Software Construction*, Prentice Hall, London, England, 1988

[101] Meyer, B., Writing Correct software, *Dr Dobbs Journal*, pp. 48 - 125, December 1989

[102] Meyer, B., *Eiffel The Language*, Prentice-Hall, London, England, 1992

[103] Miller Jnr., E. F., Methodology for Comprehensive Testing, *Tech. Rep.* , Rome Air Development Centre, Griffiss Air Force Base, New York, 1975

[104] Muralidharan, S. and Weide, B. W., Should Data Abstraction be Violated to Enhance Software Reuse ?, in *Proceedings of the 8th Annual National Conference on Ada Technology*, pp. 515 - 524, U.S. Army Communications Electronic Command, 1990

[105] Murphy, G. C. and Wong, P., *Towards a Testing Methodology for Object-Oriented Systems*, a Poster Presented at the Object-Oriented Systems, Languages and Applications Conference

[106] Myers, G. J., *The Art of Software Testing*, John Wiley, 1979

[107] *Oxford Dictionary of Current English*, 7th Ed., Oxford University Press, Oxford, England, 1984

[108] Olthoff, W. G., Augumentation of Object Oriented Programming by Concepts of Abstract Data Type Theory : The ModPascal Experience, in *Proceedings of the Object-Oriented Programming: Systems, Languages and Applications Conference*, pp. 429 - 443, ACM SIGPLAN Notices, 1986

[109] Overbeck, J., Testing Object-Oriented Software - State of the Art and Research Directions, in *Proceedings of EUROSTAR*, pp. 5/1 - 5/25, 1993

[110] Parnas, D., On the Criteria to be used in Decomposing Systems into Modules, *Communications of the ACM*, vol. 15, no. 12, pp. 1053 - 1058, December 1972

[111] Pascoe, G. A., Elements of Object-Oriented Programming, *BYTE*, pp. 15 - 20, August 1986

[112] Perry, D. E. and Kaiser, G. E., Adequate Testing and Object Oriented Programming, *Journal of Object-Oriented Programming*, pp. 13 - 19, January/Febuary 1990

[113] Phillips, N. C. K., Safe Data Type Specifications, *IEEE Transactions on Software Engineering*, vol. SE-10, no. 3, pp. 285 - 289, May 1984

[114] Pokkunuri, B. P., Object-Oriented Porgramming, *SIGPLAN Notices*, vol. 24, no. 11, pp. 96 - 101

[115] Purchase, J. A. and Winder, R. L., Debugging Tools for Object-Oriented Programming, *Tech. Rep. RN/89/77*, Dept. Computer Science, Univeristy College London, England, 1989

[116] Rayward-Smith, V. J., *A First Course in Formal Language Theory*, Blackwell Scientific Publications, London, England, 1983

[117] Rentsch, T., Object Oriented Programming, *ACM SIGPLAN Notices*, pp. 51 - 57, September 1982

[118] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modelling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991

[119] Schorre, V., A Program Verifier With Assertions in Terms of Abstract Data, in *Proceedings of the Symposium on Computer Software Engineering*, pp. 267 - 280, 1976

[120] Sciore, E., Object Specialisation, *ACM Transactions on Information Systems*, vol. 7, no. 2, pp. 103 - 122, April 1989

[121] Shlaer, S. and Mellor, S. J., *Object Lifecycles : Modeling the World in States*, Yourdon Press (Prentice-Hall), Englewood Cliffs, New Jersey, USA, 1992

[122] Smith, M. D. and Robson D. J., Object-Oriented Programming - the Problems of Validation, in *Proceedings of the 6th International Conference on Software Maintenance*, pp. 272 - 282, IEEE Computer Society Press, 1990

[123] Smith, M. D. and Robson, D. J., A Framework for Testing Object-Oriented Programs, *Journal of Object-Oriented Programming*, vol. 5, pp. 45 - 53, June 1992

[124] Snyder, A., Encapsulation And Inheritance in Object-Oriented Programming Languages, in *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 38 - 45, ACM SIGPLAN Notices, 1986

[125] Snyder, A., The Essence of Objects : Concepts and Terms, *IEEE Software*, pp. 31 - 42, January 1993

[126] Snyder, A., CommonObjects: An Overview, *SIGPLAN Notices*, vol. 21, no. 10, pp. 19 - 28, October 1986

[127] Sommerville, I., *Software Engineering*, 4th Ed., Addison Wesley, Wokingham, England, 1992

[128] Stein, L. A., Delegation Is Inheritance, in *Proceedings of Object-Oriented Programming Languages, Systems and Applications Conference*, pp. 138 - 146, ACM SIGPLAN Notices, 1987

[129] Stepney, S., Barden, R. and Cooper, D., A Survey of Object-Orientation in Z, *Software Engineering Journal*, pp. 150 - 160, March 1992

[130] Strom, R., A Comparison of the Object-Oriented and Process Paradigms, *ACM SIGPLAN Notices*, vol. 21, no. 10, pp. 88 - 97, October 1986

[131] Strooper, P. and Hoffman, D., Prolog Testing of C Modules, in *Proceedings of the International Logic Programming Symposium*, pp. 596 - 608, MIT Press, 1991

[132] Stroustrup, B., Classes: An Abstract Data Type Facility for the C Language, *ACM SIGPLAN Notices*, vol. 17, no. 1, pp. 42 - 51, January 1982

[133] Stroustrup, B., *The C++ Progrmming Language*, Addison-Wesley, Reading, Massachusetts, 1986

[134] Stroustrup, B., What Is Object Oriented Programming, *IEEE Software*, May 1988

[135] Thuy, N. N. Q, Testability and Unit tests in Large Object-Oriented Software, in *Proceedings of the Quality Week Conference*, pp. 1 - 9, 1992

[136] Trausan-Matu, S., Tepandi, J. and Barbucceanu, M., Validation and Verification of Object-Oriented Programs: Methods and Tools, in *Proceedings of EastEurOOPe 1991*, pp. 62 - 71, 1991

[137] Turner, C. D. and Robson, D.J., LCSAJ Test Coverage Tools for Modula-2, *Tech. Rep. TR 4/93*, Dept of Computer Science, Durham University, England, 1993

[138] Turner, C. D. and Robson, D. J., The Testing of Object-Oriented Programs, *Tech. Rep. TR 13/92*, Dept. of Computer Science, University of Durham, England, 1992

[139] Turner, C. D. and Robson, D. J., State-Based Testing and Inheritance, *Tech. Rep. TR 1/93*, Dept. of Computer Science, University of Durham, England, 1993

[140] Turner, C. D. and Robson, D. J., A Suite of Tools for the State-Based Testing of Object-Oriented Programs, *Tech. Rep. TR 14/92*, Dept. of Computer Science, University of Durham, England, 1992

[141] Turner, C. D. and Robson, D. J., Guidance For the Testing of Object-Oriented Programs, *Tech. Rep. TR 2/93*, Dept. of Computer Science, University of Durham, England, 1993

[142] Turner, C. D., and Robson, D. J., State-Based Testing of Object-Oriented Programs, in *Proceedings of the IEEE Conference on Software Maintenance*, pp. 302 - 310, IEEE, 1993

[143] Ungar, D. and Smith, R. B., Self : the Power of Simplicity, in *Proceedings of the Object-Oriented Systems, Languages and Applications Conference*, pp. 227 - 242, ACM SIGPLAN Notices vol. 22 no. 12, 1987

[144] United States Department Of Defense, Reference Manual for the Ada Programming Language, *Tech. Rep. MIL-STD 1815*, 1980

[145] Wallace, D. R. and Fujii, R. U., Software Verification and Validation : an Overview, *IEEE Software*, vol. 6, no. 3, pp. 10 - 17, May 1989

[146] Wegner, P., Perspectives on Object-Oriented Programming, *Tech. Rep. CS-86-25*, Department of Computer Science, Brown University, Providence, Rhode Island, USA, 1986

[147] Weyuker, E. J., How to Decide When to Stop Testing, in *Proceedings of the 5th Annual Pacific NorthWest Conference on Software Quality*, 1987

[148] Weyuker, E. J., The Evaluation of Program-Based Software Test Data Adequacy Criteria, *Communications of the ACM*, vol. 31, no. 6, pp. 668 - 675, June 1988

[149] Winfrey, T. L., Testing Object-Oriented Programs by Mutually Suspicious Parties, *Tech. Rep. CUCS-041-90*, Department of Computer Science, Columbia University, New York, 1990

[150] Wirfs-Brock, A. and Wilkerson, B., Variables Limit Reusability, *Journal of Object-Oriented Programming*, vol. 2, no. 1, pp. 34 - 42, May/June 1989

[151] Wirfs-Brock, R. J. and Johnson, R. E., Surveying Current Research in Object-Oriented Design, *Communications of the ACM*, vol. 33, no. 9, pp. 104 - 124, September 1990

[152] Wirth, N., *Programming in Modula-2*, 3rd Corrected Ed., Springer-Verlag, Berlin, Germany, 1985

[153] Wolczko, M., Ecapsulation, Delegation and Inheritance in Object-Oriented Languages, *Software Engineering Journal*, March 1992

[154] Woodfield, S. N., Gibbs, N. E. and Collofello, J. S., Improved Software Reliability Through the Use of Functional and Structural Testing, in *Proceedings of IEEE Phoenix Conference on Computers and Communications*, pp. 154 - 157, IEEE, 1983

[155] Woodward, M. R. and Hedley, D. and Hennell, M. A., Experience with Path Analysis and Testing of Programs, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 278 - 286, May 1980

12. Glossary

Acceptance testing - A testing phase applied to software very late on in the life-cycle. This type of testing is usually applied by the customer as a requirement for their acceptance of the product. Its purpose is to validate that the software built was what the customer ordered.

Assertions - Statements about a programs state at a particular point in its execution are referred to as assertions. They usually involve asserting a statement about the validity of data held in variables. Assertions applied to the parameters passed to an operation are known as preconditions. Assertions applied to the return results of an operation are known as postconditions. Statements about the current validity of an object's state are known as invariants. These are described in more detail elsewhere in this glossary.

Attributes - The name given to a variable that forms part of the data-structure used to represent the state of an object. This term is taken from the language Eiffel. In C++, they are known as data-members.

Base-class - See Parent class.

Black-box testing - Testing that is applied to a program, or unit of a program without any regard for the internal design or structure of the program. Test cases are obtained only from the programs requirements or specification. Hence, the program is treated as a 'black-box' with inputs and outputs.

Causes - These form part of a testing technique known as cause-effect graphing. A cause is a significant input sequence of values to a program or unit. Causes are combined using Boolean logic to describe the required situation for an effect to take place. Effects are described elsewhere in this glossary.

Child class - A class that has been defined by inheritance. It inherits attributes and/or operations from its parent class(es).

Class - A template from which objects are created at run-time using a class-based OOPL. It defines both the attributes that are used to represent an objects state, and the operations that are used to manipulate the state.

Cloning - Prototype-based OOPLs create objects by copying objects that already exist in the system such as prototype objects. This copying process is known as cloning.

Control-information - An object's state is generally used for two purposes, control-information and data-storage (described elsewhere in this glossary). control-information is information or values that are used to represent special situations or events. This part of the state is used to communicate events between the operations of a class.

Coupling - Coupling, or data-coupling as it is a qualitative measure of a program's interconnectivity. It is considered good software engineering practise to pass any data required by an operation as parameters therefore removing the need to access global variables. A function would therefore have no hidden side effects because all effects could be seen in the parameter values. in object-oriented programming, there is a high level of coupling within the operations of provided by an object, however there is generally low coupling between objects.

Data-storage - This is the other main use of an object's state. As its name suggests, its purpose is to simply store values that are retrieved and manipulated later. This is subtly different from control-information

because the values used in data-storage have no special meanings other than their actual values, whereas control-information stores values used to communicate situations and events to other operations. An attribute of an object can be used for either data-storage or control-information, or both.

Data abstraction - Data abstraction is the process of developing a data structure and the operations to manipulate it therefore abstracting out unnecessary information from the programmer. The data structure is manipulated via the operations so knowledge of the implementation is not needed and is usually discouraged.

data-members - Also known as attributes.

data-scenarios - Data scenarios (also known simply as scenarios) are specific situations that are fundamental to the data structure under test.

data-storage - This is a type of information store in the state of an object. Data storage information is as the name suggestions, information that has been stored, for later retrieval. It does not correspond to particular situations or event within the current class.

Data-structure - A data-structure is the result of encapsulating together into a single unit, all the data and information which corresponds to a particular task. Most languages provide a construct for performing the encapsulation, for example in C++ the `struct` construct.

Decision-decision path - A decision is a conditional branch statement within a program, hence a decision-decision path (or DD-path) is a sub-path that starts at a decision statement and ends at a decision statement.

Delegation - Delegation is an alternative code-reuse mechanism to inheritance. In languages that support delegation, an object receiving a message can delegate another object to answer it. The relationship between the two objects can usually be defined either statically (compile-time), or dynamically (runtime).

Derived class - Also known as a child class.

Domain testing - This is a specialisation of path testing. Its aim is to detect domain errors which are inputs that cause the execution to follow and invalid path.

Dynamic binding - At compile time, it is not always possible for a compiler to determine which implementation of an overloaded operation is to be called at a particular point. The binding of the call to the correct implementation must therefore be delayed until run-time. The binding is performed immediately prior to the call and hence can change dynamically depending upon the type of the object for which the call is being made.

Encapsulation - This is the grouping together into a single unit of entities, such as attributes and operations into a class.

Feature - Also known as an operation.

Finite state automaton - A mathematical model of a system whose next state is only dependent upon its current state and the sequence of its inputs.

General-substate-values - A range of values that an attribute may possess which are expected to be treated in a similar manner by the code under test and hence should not be distinguished between.

Generators - A segment of code that is used to place the object under test in a particular state. Each generator is capable of putting the object into a different state. The tool MKTC uses generators to automate the creation of states within a test case. Generators can be reused between test cases and so their use reduces the amount of code that must be written by the tester.

Generic class - Also known as a template, or meta-class. A class from which another class can be produced. They are defined using a generic unspecified type to represent the type that can change. For example a list template could be defined such that given a class as a parameter, a list of objects of that class could be created.

Grey-box testing - The combination of a white-box technique with a black-box technique is used to achieve a more effective technique. Black-box techniques cannot guarantee complete code coverage, whereas white-box techniques cannot be used to detect missing paths within code.

Hybrid OOPLs - OOPLs that have evolved from non-OOPLs. For example, C++ evolved from the language C.

Identity - This is a property of all objects. It has two components, durability and distinguishability. Durability is the ability of an object to exist between calls to its operations. Distinguishability is the property that allows programmers, or programs to tell object apart.

Implementation inheritance - This is the use of inheritance to express a code dependency between two classes, although there is no conceptual reuse. For example, inheriting from a hash-table class to provide the implementation for a binary search tree. A binary search tree *is not a* hash-table and so there is no conceptual dependency, only a code dependency.

Information hiding - The technique by which the implementation of a data-abstraction is hidden from a programmer is known as information hiding.

Inheritance - This is a facility provided by a large number of OOPLs for both code and conceptual reuse. If class B inherits from class A, then class B contains the attributes and operations of class A is addition to those defined within class B. Inherited operations can then be redefined.

Instance variables - Also known as attributes.

Instantiation - Instantiation means "to create and instance". For example, if class A is instantiated, an object is created from the class. The class is therefore a template for a particular type of object.

Integration testing - This type of testing involves the testing of objects in combination with each other. The integration is usually performed incrementally with each increment being tested separately.

Inter-class - This means within a class.

Intra-class	- This means between classes.
Invariant	- A class invariant is a declaration of the valid state that an object can possess. The declaration is usually applied to the state of an object before and after an operation has been called. It is a special type of assertion.
Member-function	- Also known as an operation.
Method	- Also known as an operation.
NULL	- The value (usually defined to be zero) used to represent an invalid pointer, that is one which does not point to a valid object.
Object	- An entity that only exists with a program at runtime. It encapsulates both state and behaviour.
Operation	- An operation is an individual segment of code that defines a particular aspect of the behaviour of an object.
Parent class	- If class B inherits from class A, class A is known as the parent class, and class B is known as the derived class.
Pointer	- Also known as a reference. A pointer is a method of addressing a particular object at runtime. Multiple pointers can point to the same object, although a pointer cannot point to multiple objects. It is usually consists of the address in the computer's memory of the object.
Postcondition	- A postcondition is an assertion that is applied when a call to an operation returns. It is used to validate that the return parameters are within their expected ranges.
Preconditions	- A precondition is an assertion that is applied when a call to an operation is made. It is used to validate that the parameters passed to the operation are within their expected ranges.
Prefixing	- This is the Simula version of inheritance. It is called prefixing because the child class declaration is prefixed by the name of the parent.

Private - An area within the class used for the declaration of attributes and operations. Any thing declared within the private area can only be accessed by the rest of the class, no other class can access this area. The name is taken from the language C++.

Protected - An area within the class used for the declaration of attributes and operations. Any thing declared within the protected area can only be accessed by the remainder of the class, and any class derived directly from it. The name is taken from the language C++.

Public - An area within the class used for the declaration of attributes and operations. As the name suggests, anything declared in this area can be accessed by any other class.

Pure OOPLs - A Pure OOPL is one which has been developed from scratch, that is, it is not based upon a non OOPL. Eiffel and Smaltalk are examples of pure OOPLs.

Runtime - Runtime is the time during which the program is executing.

Scenarios - See data scenarios.

Specific-substate-values - A specific value that an attribute may possess which is expected to be treated in a unique manner by the code under test and hence should be treated in isolation.

Specification inheritance - This is the use of inheritance for conceptual reuse. Generally, the use of specification inheritance infers that the child class is a specialisation of the parent class.

State - This is the existing condition of an object.

Static binding - At compile-time the compiler attempts to resolve all the calls to operations with the actual implementation of the operation being called. If this resolution can be performed at compile-time it is known as static-binding, otherwise it must be performed at runtime and is known as dynamic binding.

Stimulus	- A stimulus is an external event that is used as a trigger for a finite state automaton. In the context of object-oriented programs, a stimulus is the invocation of a particular operation with a unique set of parameter values.
Substate	- A substate is the name given to an attribute that is represented by its set of general and specific-substate-values. The name is used to imply it is part of the whole state of an object.
System testing	- The testing of the whole system. The system is treated as a black-box and input values are applied. Hence, system testing is not affected by the technique used to design and develop the software.
Test driver	- A test driver is a segment of code that is used to initiate a test case. Test drivers are used during unit and integration testing to simulate the program portion that would have created and invoked the operations of the objects under test.
Test oracle	- A machine or person that is capable of determining if a test case succeeded or failed.
Test stubs	- A test stub is a code segment used to simulate classes called by the current classes under test. Stubs are used when the classes required are either too complex to be used during testing, or have not been written yet.
Unit testing	- The process of testing the units in isolation that the program is constructed from. Testing classes in isolation usually requires the use of test stubs and drivers.
White-box	- White-box testing techniques are a category of testing techniques that use information derived from the program structure and code to determine the test cases created and executed.

Appendix 1

The Class ilist

The implementation for the class ilist:

```
1   /* a singly linked list of integers
2    * the order is determined by the order in which they were inserted
3    * Author C.D. Turner
4    * Date 3/6/92
5    */
6
7   #include "ilist.hpp"
8
9   Slink::Slink (int pnObject)
10  {
11      pnItem = pnObject;
12      plNext = NULL;
13  }
14
15  void Slink::item(int pnObject)
16  {
17      if (pnObject != pnItem)
18      {
19          pnItem = pnObject;
20      }
21  }
22
23  int Slink::item() const
24  {
25      return pnItem;
26  }
27
28  CDT_Slist::CDT_Slist()
29  {
30      pTop = NULL;
31      pCur = &pTop;
32  }
33
34  CDT_Slist::CDT_Slist(const CDT_Slist & Copy)
35  {
36      pTop = NULL;
37      pCur = &pTop;
38
39      // pass the buck to the assignment operator
40      *this = Copy;
41  }
42
43  CDT_Slist::~CDT_Slist()
```

```
44  {
45      clear();
46  }
47
48  void CDT_Slist::clear()
49  {
50      Slink * pTmp = pTop;
51      Slink * pNextLink;
52
53      while (pTmp != NULL)
54      {
55          pNextLink = pTmp->plNext;
56          delete pTmp;
57          pTmp = pNextLink;
58      }
59
60      pTop = NULL;
61      pCur = &pTop;
62  }
63
64  BOOL CDT_Slist::prepend(int pnObject)
65  {
66      Slink * plNew = new Slink(pnObject);
67      if (plNew == NULL)
68          return FALSE;
69
70      plNew->plNext = pTop;
71      pTop = plNew;
72      return TRUE;
73  }
74
75  BOOL CDT_Slist::append(int pnObject)
76  {
77      Slink * plNew = new Slink(pnObject);
78      if (plNew == NULL)
79          return FALSE;
80
81      if (pTop == NULL)
82      {
83          pTop = plNew;
84          pTop->plNext = NULL;
85      }
86      else
87      {
88          Slink * pTmp = pTop;
89          while (pTmp->plNext != NULL)
90              pTmp = pTmp->plNext;
91
92          pTmp->plNext = plNew;
93          plNew->plNext = NULL;
94      }
95      return TRUE;
96  }
97
98  BOOL CDT_Slist::find(int pnObject) const
99  {
100     Slink * pTmp = pTop;
101
102     while (pTmp != NULL)
103     {
104         if (pnObject == pTmp->item())
105             return TRUE;
106         else
107             pTmp=pTmp->plNext;
108     }
109     return FALSE;
110 }
111
112 BOOL CDT_Slist::remove(int pnObject)
113 {
114     Slink * pTmp = pTop;
115     Slink * pLast = NULL;
116
117     while (pTmp != NULL)
118     {
119         if (pTmp->item() == pnObject)
120         {
121             if (pLast == NULL)
122             {
123                 pTop = pTmp->plNext;
```

```
124                 delete pTmp;
125                 pCur = &pTop;
126             }
127             else
128             {
129                 pLast->plNext = pTmp->plNext;
130                 delete pTmp;
131                 pCur = &pTop;
132             }
133             return TRUE;
134         }
135         else
136         {
137             pLast = pTmp;
138             pTmp = pTmp->plNext;
139         }
140     }
141     return FALSE;
142 }
143
144 BOOL CDT_Slist::replace(int pnOriginal, int pnNew)
145 {
146     Slink *pTmp = pTop;
147
148     while (pTmp != NULL)
149     {
150         if (pnOriginal == pTmp->item())
151         {
152             pTmp->item(pnNew);
153             return TRUE;
154         }
155         else
156             pTmp = pTmp->plNext;
157     }
158     return FALSE;
159 }
160
161 BOOL CDT_Slist::first()
162 {
163     pCur = &pTop;
164     if (*pCur == NULL)
165         return FALSE;
166     else
167         return TRUE;
168 }
169
170 BOOL CDT_Slist::next()
171 {
172     if (*pCur == NULL)
173         return FALSE;
174     else
175     {
176         if ((*pCur)->plNext == NULL)
177         {
178             pCur = &((*pCur)->plNext);
179             return FALSE;
180         }
181         else
182         {
183             pCur = &((*pCur)->plNext);
184             return TRUE;
185         }
186     }
187 }
188
189 BOOL CDT_Slist::get(int & pnObject) const
190 {
191     if (*pCur == NULL)
192         return FALSE;
193     else
194     {
195         pnObject = (*pCur)->item();
196         return TRUE;
197     }
198 }
199
200 BOOL CDT_Slist::set(int pnNew)
201 {
202     if (*pCur == NULL)
203         return FALSE;
```

```
204         else
205         {
206             (*pCur)->item(pnNew);
207             return TRUE;
208         }
209     }
210
211     BOOL CDT_Slist::insert(int pnObject)
212     {
213         Slink *pTmp = new Slink(pnObject);
214         if (pTmp == NULL)
215             return FALSE;
216         pTmp->plNext = *pCur;
217         *pCur = pTmp;
218         return TRUE;
219     }
220
221     BOOL CDT_Slist::remove()
222     {
223         if (*pCur == NULL)
224             return FALSE;
225         else
226         {
227             Slink *pTmp = *pCur;
228             *pCur = pTmp->plNext;
229             delete pTmp;
230             return TRUE;
231         }
232     }
233
234     int CDT_Slist::size() const
235     {
236         Slink * pTmp = pTop;
237         int iCount = 0;
238
239         while (pTmp != NULL)
240         {
241             iCount ++;
242             pTmp = pTmp->plNext;
243         }
244         return iCount;
245     }
246
247     int CDT_Slist::toend() const
248     {
249         if (*pCur == NULL)
250             return -1;
251
252         Slink * pTmp = *pCur;
253         int iCount = 0;
254
255         while (pTmp != NULL)
256         {
257             iCount ++;
258             pTmp = pTmp->plNext;
259         }
260         return iCount - 1;
261     }
262
263     BOOL CDT_Slist::operator==(const CDT_Slist & Compare) const
264     {
265         if (size() != Compare.size())
266             return FALSE;
267
268         CDT_Slist lCopy(Compare);
269
270         Slink * pTmp = pTop;
271         for (int iLoop = 1; iLoop <= Compare.size(); iLoop ++)
272         {
273             if (lCopy.remove(pTmp->item()) == FALSE)
274                 return FALSE;
275             pTmp = pTmp->plNext;
276         }
277         return TRUE;
278     }
279
280     BOOL CDT_Slist::operator!=(const CDT_Slist & Compare) const
281     {
282         return !(this->operator==(Compare));
283     }
```

```
284
285     void CDT_Slist::operator=(const CDT_Slist & Copy)
286     {
287         clear();
288         Slink * pTmp = Copy.pTop;
289         while(pTmp != NULL)
290         {
291             append(pTmp->item());
292             pTmp = pTmp->plNext;
293         }
294     }
```

Appendix 2

The Test Control Script for the class ilist

```
# test definition file for the singly linked
# list template class. It has been instatiated
# to form a list of integers for the purpose of testing

TEST ENGINEER      "Chris Turner"
CLASS UNDER TEST   Tilist
DATE               "03/08/92"
INCLUDE FILES =    "Tilist.hpp", "markers.h";

SUBSTATES:
    NUMBER OF SUBSTATES = 5
    # substate 1 is a reflection of the value of pTop
    # 1: pTop = Null  : value 1
    # 2: pTop != Null : value 2
    SUBSTATE 1 HAS 2 VALUES

    # substate 2 is a reflection of the value of *pCur
    # 1:*pCur = NULL
    # 2: *pCur != Null
    SUBSTATE 2 HAS 2 VALUES

    # substate 3 is a reflection of the value of pCur
    # 1: pCur = &pTop
    # 2: pCur != &pTop
    SUBSTATE 3 HAS 2 VALUES

    # substate 4 is a reflection of the value of pCur->plNextLink
    # 1: *pCur->plNext = NULL
    # 2: *pCur->plNext != Null
    # 3: undefined (*pCur = NULL)
    SUBSTATE 4 HAS 3 VALUES

    #substate 5 is a conditional test based on the value of pCur->item()
    # it will be used soley for s[5]($) to test for a change in value
    # 1: (*pCur)->item() is valid
    # 2: (*pCur)->item() is invaild (ie s[2](1))
    SUBSTATE 5 HAS 2 VALUES
END

DATA SCENARIOS:
    NUMBER OF SCENARIOS = 6
    # scenario 1 is an empty list, pCur = pTop
    SCENARIO 1 : S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2);
```

```
    #scenario 2 is a single element list, pCur = pTop
    SCENARIO 2 : S[1](2) & S[2](2) & S[3](1) & S[4](1) & S[5](1);

    #scenario 3 is a single element list, pCur past end of list
    SCENARIO 3 : S[1](2) & S[2](1) & S[3](2) & S[4](3) & S[5](2);

    #scenario 4 is a two element list with pCur = pTop
    SCENARIO 4 : S[1](2) & S[2](2) & S[3](1) & S[4](2) & S[5](1);

    #scenario 5 is a two element list with pCur between two elements
    SCENARIO 5 : S[1](2) & S[2](2) & S[3](2) & S[4](1) & S[5](1);

    #scenario 6 is a three element list with pCur = middle element
    SCENARIO 6 : S[1](2) & S[2](2) & S[3](2) & S[4](2) & S[5](1);
END

INVARIANT:
   FEATURE = invariant
       S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2)
   |    S[1](2) & S[2](1) & S[3](2) & S[4](3) & S[5](2)
   |    S[1](2) & S[2](2) & S[3](1) & !S[4](3) & S[5](1)
   |    S[1](2) & S[2](2) & S[3](2) & !S[4](3) & S[5](1);
END

GENERATORS:
    NAME = Generator1
    PRECONDITION = NONE
#    POSTCONDITION = S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2);
    POSTCONDITION = SCENARIO 1;
    GENERATOR
    {{
        markerson();
        MKTC_Object = new Tilist;
    }}
    TERMINATOR
    {{
        if (MKTC_Object != NULL)
            delete MKTC_Object;
        markersoff();
    }}

    NAME = Generator2
    PRECONDITION = NONE
    # postcondition is S[1](2) & S[2](2) & S[3](1) & S[4](1) & S[5](1);
    POSTCONDITION = SCENARIO 2;
    GENERATOR
    {{
        MKTC_Object = new Tilist;
        test(MKTC_Object->append(1) == TRUE);
        test(MKTC_Object->size() == 1);

        test(MKTC_Object->first() == TRUE);
    }}
    TERMINATOR
    {{
        if (MKTC_Object != NULL)
            delete MKTC_Object;
    }}

    NAME = Generator3
    # PRECONDITION Generator 1
    PRECONDITION = S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2);
    # postcondition is S[1](2) & S[2](1) & S[3](2) & S[4](3) & S[5](2);
    POSTCONDITION = SCENARIO 3;
    GENERATOR
    {{
        // create a non empty list
        test(MKTC_Object->append(1) == TRUE);
        test(MKTC_Object->size() == 1);

        // now move the current pointer past the end of the list
        test(MKTC_Object->first() == TRUE);
        test(MKTC_Object->next() == FALSE);
    }}

    NAME = Generator4
    # PRECONDITION Generator 1
    PRECONDITION = S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2);
    # postcondition is S[1](2) & S[2](2) & S[3](1) & S[4](2) & S[5](1);
```

```
    POSTCONDITION = SCENARIO 4;
    GENERATOR
    {{
        // store two elements in the list
        test(MKTC_Object->append(1) == TRUE);
        test(MKTC_Object->append(30) == TRUE);
        test(MKTC_Object->size() > 1);

        // now move the s[3](1 -> 1) & s[4](2->2)
        test(MKTC_Object->first() == TRUE);
    }}

    NAME = Generator5
    # PRECONDITION Generator 1
    PRECONDITION = S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2);
    # postcondition is S[1](2) & S[2](2) & S[3](2) & S[4](1) & S[5](1);
    POSTCONDITION = SCENARIO 5;
    GENERATOR
    {{
        // store two elements in the list
        test(MKTC_Object->append(1) == TRUE);
        test(MKTC_Object->append(30) == TRUE);
        test(MKTC_Object->size() > 1);

        // now move the s[3](1 -> 2) & s[4](2->1)
        test(MKTC_Object->first() == TRUE);
        test(MKTC_Object->next() == TRUE);
    }}

    NAME = Generator6
    PRECONDITION = S[1](1) & S[2](1) & S[3](1) & S[4](3) & S[5](2);
    # postcondition is S[1](2) & S[2](2) & S[3](2) & S[4](2) & S[5](1);
    POSTCONDITION = SCENARIO 6;
    GENERATOR
    {{
        // store two elements in the list
        test(MKTC_Object->append(1) == TRUE);
        test(MKTC_Object->append(30) == TRUE);
        test(MKTC_Object->append(3) == TRUE);
        test(MKTC_Object->size() > 1);

        // now move the s[3](1 -> 2) & s[4](2->2)
        test(MKTC_Object->first() == TRUE);
        test(MKTC_Object->next() == TRUE);
    }}
END

SUBSTATE TESTS:
    SUBSTATE = 1
    FEATURE = TestSubstate1
    CHANGE = TestSubstate1Change
    SUBSTATE = 2
    FEATURE = TestSubstate2
    CHANGE = TestSubstate2Change
    SUBSTATE = 3
    FEATURE = TestSubstate3
    CHANGE = TestSubstate3Change
    SUBSTATE = 4
    FEATURE = TestSubstate4
    CHANGE = TestSubstate4Change
    SUBSTATE = 5
    FEATURE = TestSubstate5
    CHANGE = TestSubstate5Change
END

TESTS:
#    FEATURE = constructor ##########################################################
#   no explicit tests are required for this, as
#    the generators should test it

    FEATURE = clear
    CASE:
        SUBSTATE = S[1](* -> 1) &
                   S[2](* -> 1) &
                   S[3](* -> 1) &
                   S[4](* -> 3) &
                   S[5](* -> 2);
        CODE
        {{
            markerson();
```

```
        Object->clear();
        markersoff();
    }}

FEATURE = append ############################################################
CASE:
    # check transition empty list -> non empty list
    SUBSTATE = SCENARIO 1 ->
               S[1](2);

    # check that append does indeed add onto the last node
    # with pCur on last node ..
    SUBSTATE = S[1](2 -! $) &
               S[2](2 -! $) &
               S[4](1 -> 2);
    # and past the last node ..
    SUBSTATE = S[1](2 -! $) &
               S[2](1 -> 2) &
               S[5](2 -> 1);

    # check that if in the middle of the list, then adding a node
    # on the end changes nothing ..
    SUBSTATE = S[1](2 -! $) &
               S[2](2 -! $) &
               S[3](2 -! $) &
               S[4](2 -! $) &
               S[5](1 -! $);
    CODE
    {{
        markerson();
        // add an arbitrary number in to the list
        test(Object->append(40) == TRUE);
        markersoff();
    }}
CASE:
    # out of memory  (no change to current list)
    SUBSTATE = S[1](* -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);
    CODE
    {{
        markerson();
        bSimOutOfMemory = true;
        // add an arbitrary number in to the list
        test(Object->append(40) == TRUE);
        markersoff();
    }}

FEATURE = prepend ###########################################################
CASE:
    # check transition empty list -> non empty list
    SUBSTATE = SCENARIO 1 ->
               S[1](2);

    # check that append does indeed add onto the first node
    # with pCur not on first node ..
    SUBSTATE = S[1](2 -> $) &
               S[2](2 -! $) &
               S[3](2 -! $) &
               S[4](1 -! $);
    # and on the first node ..
    SUBSTATE = S[1](2 -> $) &
               S[2](* -> $) &
               S[3](1 -! $);

    # check that if in the middle of the list, then adding a node
    # on the front changes nothing .. except the pTop
    SUBSTATE = S[1](2 -> $) &
               S[2](2 -! $) &
               S[3](2 -! $) &
               S[4](2 -! $) &
               S[5](1 -! $);
    CODE
    {{
        markerson();
        // add an arbitrary number in to the list
        test(Object->prepend(40) == TRUE);
        markersoff();
```

```
    }}
CASE:
    # out of memory  (no change to current list)
    SUBSTATE = S[1](* -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);
    CODE
    {{
        markerson();
        bSimOutOfMemory = true;
        // add an arbitrary number in to the list
        test(Object->prepend(40) == TRUE);
        markersoff();
    }}

FEATURE = find ##################################################################
CASE:
    # test for a element that doesnt exist in the list
    # and test for an element in an empty list (S[1](1) ..)
    SUBSTATE = S[1](* -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);
    CODE
    {{
        markerson();
        test(Object->find(99) == FALSE);
        markersoff();
    }}

CASE:
    # test for the first element in the list
    SUBSTATE = S[1](2 -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);
    CODE
    {{
        markerson();
        test(Object->find(1) == TRUE);
        markersoff();
    }}

CASE:
    ## check for a.n.other element in the list
    SUBSTATE = SCENARIO 4..6 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);
    CODE
    {{
        markerson();
        test(Object->find(30) == TRUE);
        markersoff();
    }}

FEATURE = remove ################################################################
CASE:
    # try and remove an element that is not in the list
    # also try it on an empty list
    SUBSTATE = S[1](* -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);

    CODE
    {{
        markerson();
        test(Object->remove(99) == FALSE);
        markersoff();
    }}

CASE:
```

```
    #remove the first element in the list
    SUBSTATE = S[1](2 -> 1) &
               S[2](2 -> $) &
               S[3](? -> 1);

    SUBSTATE = S[1](2 -> 1) &
               S[2](1 -> 1) &
               S[3](* -> 1);
    CODE
    {{
        markerson();
        test(Object->remove(1) == TRUE);
        markersoff();
    }}

CASE:
    # remove any element from the list ..
    SUBSTATE = SCENARIO 4..6 ->
               S[1](2) &
               S[2](2) &
               S[3](1);
    CODE
    {{
        markerson();
        test(Object->remove(30) == TRUE);
        markersoff();
    }}

FEATURE = replace ############################################################
CASE:
    # x isnt in either an empty list or a non empty list
    SUBSTATE = S[1](* -! $) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);
    CODE
    {{
        markerson();
        test(Object->replace(99, 57) == FALSE);
        markersoff();
    }}

CASE:
    # replace an element (1) that is in the list
    SUBSTATE = S[1](2 -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $);
    CODE
    {{
        markerson();
        test(Object->replace(1, 56) == TRUE);
        markersoff();
    }}

CASE:
    # replace an element (30) that is in the list
    SUBSTATE = SCENARIO 4..6 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($);
    CODE
    {{
        markerson();
        test(Object->replace(30, 56) == TRUE);
        markersoff();
    }}

FEATURE = first ##############################################################
CASE:
    # on an empty list
    SUBSTATE = SCENARIO 1 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);
    CODE
```

```
    {{
        markerson();
        test(Object->first() == FALSE);
        markersoff();
    }}
CASE:
    # on a non empty list ..
    SUBSTATE = S[1](2 -! $) &
               S[2](* -> 2) &
               S[3](* -> 1);
    CODE
    {{
        markerson();
        test(Object->first() == TRUE);
        markersoff();
    }}


FEATURE = next ###################################################################
CASE:
    # empty list
    SUBSTATE = SCENARIO 1 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);

    # past the end of the list
    SUBSTATE = S[1](2 -! $) &
               S[2](1 -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);

    # at the end of the list
    SUBSTATE = S[1](2 -! $) &
               S[2](2 -> 1) &
               S[3](* -> 2) &
               S[4](1 -> 3) &
               S[5](1 -> 2);

    # at the end of the list
    SUBSTATE = SCENARIO 5 ->
               !S[1]($) &
               S[2](1) &
               S[3](2) &
               S[4](1) &
               S[5]($);
    CODE
    {{
        markerson();
        test(Object->next() == FALSE);
        markersoff();
    }}

CASE:
    # not at the end of the list
    SUBSTATE = SCENARIO 4 ->
               !S[1]($) &
               S[2](2) &
               S[3](2) &
               S[4](1) &
               S[5]($);

    CODE
    {{
        markerson();
        test(Object->next() == TRUE);
        markersoff();
    }}

FEATURE = get ####################################################################
CASE:
    # on an empty list ..
    SUBSTATE = SCENARIO 1 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
```

```
            !S[5](3);
    CODE
    {{
        markerson();
        int iArg;
        test(Object->get(iArg) == FALSE);
        markersoff();
    }}

CASE:
    # on all possibilites of a none empty list with a valid current item
    SUBSTATE = S[1](2 -! 3) &
               S[2](* -! 3) &
               S[3](1 -! 3) &
               S[4](* -! 3) &
               S[5](1 -! 3);
    CODE
    {{
        markerson();
        int iArg;
        test(Object->get(iArg) == TRUE);
        test(iArg == 1);
        markersoff();
    }}

CASE:
    # on all possibilites of a none empty list with a valid current item
    SUBSTATE = S[1](2 -! 3) &
               S[2](* -! 3) &
               S[3](2 -! 3) &
               S[4](* -! 3) &
               S[5](1 -! 3);
    CODE
    {{
        markerson();
        int iArg;
        test(Object->get(iArg) == TRUE);
        test(iArg == 30);
        markersoff();
    }}

CASE:
    # on all possibilites of a none empty list with an invalid cur item
    SUBSTATE = S[1](2 -! 3) &
               S[2](* -! 3) &
               S[3](* -! 3) &
               S[4](* -! 3) &
               S[5](2 -! 3);
    CODE
    {{
        markerson();
        int iArg;
        test(Object->get(iArg) == FALSE);
        markersoff();
    }}

FEATURE = set ##############################################################
CASE:
    # an empty list ...
    SUBSTATE = SCENARIO 1 ->
               !S[1](3) &
               !S[2](3) &
               !S[3](3) &
               !S[4](3) &
               !S[5](3);
    CODE
    {{
        markerson();
        test(Object->set(99) == FALSE);
        markersoff();
    }}

CASE:
    # all possilbilites of a valid list with a valid cur item
    SUBSTATE = S[1](2 -! 3) &
               S[2](* -! 3) &
               S[3](* -! 3) &
               S[4](* -! 3) &
               S[5](1 -> 3);
    CODE
```

```
    {{
        markerson();
        test(Object->set(99) == TRUE);
        markersoff();
    }}

CASE:
    # all possilbilites of a valid list with an invalid cur item
    SUBSTATE = S[1](2 -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](2 -! $);
    CODE
    {{
        markerson();
        test(Object->set(99) == FALSE);
        markersoff();
    }}

FEATURE = insert ###############################################################
CASE:
    # On an empty list ...
    SUBSTATE = SCENARIO 1 ->
               S[1]($) &
               S[2]($) &
               !S[3]($);

    # before the first element of the list ..
    SUBSTATE = SCENARIO 2,4 ->
               S[1]($) &
               S[2]($) &
               !S[3]($) &
               S[4]($);

    # after the end of the list
    SUBSTATE = SCENARIO 3 ->
               !S[1]($) &
               S[2]($) &
               !S[3]($) &
               S[4](1) &
               S[5](1);

    # in the middle of the lsit somewhere ...
    SUBSTATE = SCENARIO 5,6 ->
               !S[1]($) &
               S[2]($) &
               !S[3]($) &
               S[4](2) &
               S[5](1);
    CODE
    {{
        markerson();
        test(Object->insert(55) == TRUE);
        markersoff();
    }}
CASE:
    # out of memory  (no change to current list)
    SUBSTATE = S[1](* -! $) &
               S[2](* -! $) &
               S[3](* -! $) &
               S[4](* -! $) &
               S[5](* -! $);
    CODE
    {{
        markerson();
        bSimOutOfMemory = true;
        // add an arbitrary number in to the list
        test(Object->insert(40) == TRUE);
        markersoff();
    }}

FEATURE = remove ###############################################################
CASE:
    # on an empty list ..
    SUBSTATE = SCENARIO 1 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
```

```
            !S[5]($);

    # past the end of the list ..
    SUBSTATE = SCENARIO 3 ->
            !S[1]($) &
            !S[2]($) &
            !S[3]($) &
            !S[4]($) &
            !S[5]($);
    CODE
    {{
        markerson();
        test(Object->remove() == FALSE);
        markersoff();
    }}

CASE:
    # on the only element in a list ..
    SUBSTATE = SCENARIO 2 ->
            S[1](1) &
            S[2](1) &
            S[3](1);

    # on the first element in a list ..
    SUBSTATE = SCENARIO 4 ->
            S[1](1) &
            S[2](2) &
            S[3](1);

    # on the last element ..
    SUBSTATE = SCENARIO 5 ->
            !S[1]($) &
            S[2](1) &
            !S[3]($) &
            S[4](3) &
            S[5](2);

    # on an element in the middle of a list ..
    SUBSTATE = SCENARIO 6 ->
            !S[1]($) &
            S[2]($) &
            !S[3]($) &
            S[4](1) &
            S[5](1);

    CODE
    {{
        markerson();
        test(Object->remove() == TRUE);
        markersoff();
    }}

FEATURE = size ################################################################
CASE:
    # empty list ..
    SUBSTATE = SCENARIO 1 ->
            !S[1]($) &
            !S[2]($) &
            !S[3]($) &
            !S[4]($) &
            !S[5]($);
    CODE
    {{
        markerson();
        test(Object->size() == 0);
        markersoff();
    }}

CASE:
    # non empty list ..
    SUBSTATE = SCENARIO 2..6 ->
            !S[1]($) &
            !S[2]($) &
            !S[3]($) &
            !S[4]($) &
            !S[5]($);
    CODE
    {{
        markerson();
        test(Object->size() > 0);
```

```
        markersoff();
    }}

FEATURE = toend ############################################################
CASE:
    # empty list ..
    # or past end of list
    SUBSTATE = SCENARIO 1,3 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);
    CODE
    {{
        markerson();
        test(Object->toend() == -1);
        markersoff();
    }}

CASE:
    # one away from end  ..
    SUBSTATE = SCENARIO 2,5 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);
    CODE
    {{
        markerson();
        test(Object->toend() == 0);
        markersoff();
    }}

CASE:
    # two away from end ..
    SUBSTATE = SCENARIO 4,6 ->
               !S[1]($) &
               !S[2]($) &
               !S[3]($) &
               !S[4]($) &
               !S[5]($);
    CODE
    {{
        markerson();
        test(Object->toend() == 1);
        markersoff();
    }}

FEATURE = assignment #######################################################
CASE:
    # the object on the left hand side takes the state of the object
    # on the right hand side ( if pCur = Top)
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = S[1](* -! $) &
                   S[2](* -! $) &
                   S[3](1 -! $) &
                   S[4](* -! $) &
                   S[5](* -! $);
        OBJECT = LeftHandSide
        SUBSTATE = S[1](* -> RightHandSide.START) &
                   S[2](* -> RightHandSide.START) &
                   S[3](* -> RightHandSide.START) &
                   S[4](* -> RightHandSide.START) &
                   S[5](* -> RightHandSide.START);
    END

    # the object on the left hand side takes the same state for
    # S[1] but not the rest if !S[2](1)
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = S[1](* -! $) &
                   S[2](* -! $) &
                   S[3](2 -! $) &
                   S[4](* -! $) &
                   S[5](* -! $);
        OBJECT = LeftHandSide
        SUBSTATE = S[1](* -> RightHandSide.START) &
```

```
                    S[2](* -> 2) &
                    S[3](* -> 1) &
                    S[5](* -> 1);
    END

    CODE
    {{
        markerson();
        *LeftHandSide = *RightHandSide;
        markersoff();
    }}
CASE:
    # out of memory
    # the object on the left hand side takes the state of the object
    # on the right hand side ( if pCur = Top)
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = S[1](* -! $) &
                   S[2](* -! $) &
                   S[3](* -! $) &
                   S[4](* -! $) &
                   S[5](* -! $);
        OBJECT = LeftHandSide
        # empty list
        SUBSTATE = S[1](* -> 1) &
                   S[2](* -> 1) &
                   S[3](* -> 1) &
                   S[4](* -> 3) &
                   S[5](* -> 2);
    END
    CODE
    {{
        markerson();
        *LeftHandSide = *RightHandSide;
        markersoff();
    }}

FEATURE = equality_test ##################################################
CASE:
    # two empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 1 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 1 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
    END

    # two none lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 2 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 2 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
    END

    # two none empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 4 ->
                   !S[1]($) &
```

```
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 4 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
    END

    # two non empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 6 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 6 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
    END
    CODE
    {{
        markerson();
        test(LeftHandSide->operator==(*RightHandSide));
        markersoff();
    }}

CASE:
    # an empty list and a non empty list ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 1 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 2.. 6 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
    END

    # two non empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 4 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 6 ->
                !S[1]($) &
                !S[2]($) &
                !S[3]($) &
                !S[4]($) &
                !S[5]($);
    END
    CODE
    {{
        markerson();
        test((LeftHandSide->operator==(*RightHandSide)) == FALSE);
        markersoff();
    }}
```

```
FEATURE = inequality_test ##################################################
CASE:
    # two empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 1 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 1 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
    END

    # two none lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 2 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 2 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
    END

    # two none empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 4 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 4 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
    END

    # two non empty lists ..
    MULTIPLE:
        OBJECT = RightHandSide
        SUBSTATE = SCENARIO 6 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
        OBJECT = LeftHandSide
        SUBSTATE = SCENARIO 6 ->
                   !S[1]($) &
                   !S[2]($) &
                   !S[3]($) &
                   !S[4]($) &
                   !S[5]($);
    END
    CODE
    {{
        markerson();
        test((LeftHandSide->operator!=(*RightHandSide)) == FALSE);
        markersoff();
    }}
```

```
    CASE:
        # an empty list and a non empty list ..
        MULTIPLE:
            OBJECT = RightHandSide
            SUBSTATE = SCENARIO 1 ->
                       !S[1]($) &
                       !S[2]($) &
                       !S[3]($) &
                       !S[4]($) &
                       !S[5]($);
            OBJECT = LeftHandSide
            SUBSTATE = SCENARIO 2.. 6 ->
                       !S[1]($) &
                       !S[2]($) &
                       !S[3]($) &
                       !S[4]($) &
                       !S[5]($);
        END

        # two non empty lists ..
        MULTIPLE:
            OBJECT = RightHandSide
            SUBSTATE = SCENARIO 4 ->
                       !S[1]($) &
                       !S[2]($) &
                       !S[3]($) &
                       !S[4]($) &
                       !S[5]($);
            OBJECT = LeftHandSide
            SUBSTATE = SCENARIO 6 ->
                       !S[1]($) &
                       !S[2]($) &
                       !S[3]($) &
                       !S[4]($) &
                       !S[5]($);
        END
        CODE
        {{
            markerson();
            test((LeftHandSide->operator!=(*RightHandSide)) == TRUE);
            markersoff();
        }}

END
```

Appendix 3

The Class string

The implementation of the class string:

```
1   #include <stddef.h>
2   #include <ctype.h>
3   #include <iostream.h>
4   #define _STRINGCPP
5   char cCrud = '\0';
6   #include "strings.hpp"
7
8   /* the implementation of the string class
9    * Author C.D. Turner
10   * Creation Date : 16/1/91
11   * Revision History :
12   * 17/1/91 -- all strings not initialised, must return zero with length()
13   * 21/7/91 -- added extended class
14   * 28/2/91 -- Corrected the errors, especially in the constructors
15   */
16  /* Details about the Data Representation:
17   * iSize            -- this is the maximum size of string that can be stored
18   *                     in the current memory block (sptr).
19   * sptr             -- this points to the memory where the string is stored.
20   * cEnd             -- a single character that is pointed to when it is
21   *                     a null object.
22   *
23   * the memory block allocated has an extra element, to store a \0 at the end
24   * if neccesary, thus the user doesn't have to worry about not having
25   * enough space.
26   * whereever possible the routines try to use the space that is already
27   * allocated rather than allocate a new block and copy to it.
28
29   * NB single line statement blocks have been expanded with braces
30   * to improve the readability when referring to line number directly
31   * This has been done for the TESTING */
32
33  const string NullString;
34  bool bSimOOM = false; // simulate out of memory errors
35
36  /*******************protected features*****************/
37  ////////////////create()////////////////////////////////
38  bool string::create(int iNewSize, char ** psNew)
39  {
40      char * cpTmp = (bSimOOM == true) ? NULL : (new char [iNewSize + 1]);
41
42      if (cpTmp != NULL)
43      {
```

```
44          cpTmp[iNewSize] = cpTmp[0] = '\0';
45          *psNew = cpTmp;
46          return true;
47      }
48      else
49      {
50          *psNew = &cEnd;
51          cEnd = '\0';
52          return false;
53      }
54  }
55
56  ////////////////destroy()////////////////////////////////
57  void string::destroy(char * cpOld, int iSize)
58  {
59      if (iSize != 0)
60      {
61          delete [] cpOld;
62      }
63  }
64
65  ////////////////////////////string(void)////////////////////////////////////
66  string::string(void)
67  {
68      iSize = 0;
69      cEnd = '\0';
70      sptr = &cEnd;
71  }
72
73  ////////////////////////////////string(string)//////////////////////////////
74  string::string(const string & sString)
75  {
76      cEnd = '\0';
77      iSize = 0;
78      sptr = &cEnd;
79
80          // pass the buck to the assignment routine
81      *this = sString;
82  }
83
84  ////////////////// construct with just a size ////////////////////////////
85  string::string(int iLen)
86  {
87      cEnd = '\0';
88
89      if (create(iLen, &sptr) == false)
90      {
91          iSize = 0;
92      }
93      else
94      {
95          iSize = iLen;
96          memset(sptr, 0, iLen);
97      }
98  }
99
100
101 ////////////////////////////string(char *)//////////////////////////////////////
102 string::string(const char * sInit)
103 {
104     cEnd = '\0';
105     int iPassedSize = strlen(sInit);
106     if (iPassedSize > 0)
107     {
108         if (create(iPassedSize, &sptr) == true)
109         {
110             strcpy(sptr, sInit);
111             iSize = iPassedSize;
112         }
113         else
114         {
115             iSize = 0;
116         }
117     }
118     else
119     {
120         iSize = 0;
121         sptr = &cEnd;
122     }
123 }
```

```
124
125
126     ////////////////// assign a char array to a string ////////////////////////
127     void string::operator=(const string & sOperand)
128     {
129         int iLen = sOperand.length();
130         if (iLen > iSize)
131         {
132             destroy(sptr, iSize);
133             if (create(iLen, &sptr) == false)
134             {
135                 iSize = 0;
136                 return;
137             }
138             else
139             {
140                 iSize = iLen;
141             }
142         }
143         if (iLen != 0)
144         {
145             strcpy(sptr, sOperand.sptr);
146         }
147         else if (iSize != 0)
148         {
149             *sptr = '\0';
150         }
151     }
152
153
154     /////////////// concatenation operator (str1 += char *;) ////////////////
155     void string::operator+=(const string & sOperand)
156     {
157         bool bNew = false;
158         int iLen = length() + sOperand.length();
159         char * sTmp;
160
161         if (iLen > iSize)
162         {
163             bNew = true;
164             if (create(iLen, &sTmp) == false)
165             {
166                     // leave it as it was.
167                 return;
168             }
169
170             strcpy(sTmp, sptr);
171         }
172         else
173         {
174             sTmp = sptr;
175         }
176         strcat(sTmp, sOperand.sptr);
177
178             // if a new store was created, then delete old one + tidyup
179         if (bNew == true)
180         {
181             destroy(sptr, iSize);
182             iSize = iLen;
183             sptr = sTmp;
184         }
185     }
186
187
188     ////////// concatenation operator (str1 = str2 + char*;) ////////////////
189     string string::operator+(const string & sOperand) const
190     {
191         string sTmp(length() + sOperand.length());
192         sTmp = *this;
193         sTmp += sOperand;
194         return sTmp;
195     }
196
197     ////////////////////////////find(char, int, bool)////////////////////////
198     int string::find(int cChar, int iStart, bool bDirection) const
199     {
200         int iPos = iStart,
201             iDir,
202             iLen = this->length();
203
```

```
204         if (bDirection == true)
205         {
206             iDir = 1;
207         }
208         else
209         {
210             iDir = -1;
211         }
212
213         if (iStart > iLen)
214         {
215             return -1;
216         }
217
218         while (iPos >= 0 && iPos <= iLen)
219         {
220             if (this->operator[](iPos) == cChar)
221             {
222                 return iPos;
223             }
224             iPos += iDir;
225         }
226                 // if reached here, must have failed the search
227         return -1;
228     }
229
230
231     ////////////////////// substring: left len chars ///////////////////////////
232     string string::left(int iLen) const
233     {
234         if (iLen <= 0)
235         {
236             return NullString;
237         }
238         string sTmp(iLen+1);
239         strncpy(sTmp.sptr, sptr, iLen);
240         sTmp[iLen] = '\0';
241         return sTmp;
242     }
243
244     ////////////////// substring: right len chars ////////////////////////////
245     string string::right(int iLen) const
246     {
247         if (iLen > length())
248         {
249             return NullString;
250         }
251         else if (iLen <= 0)
252         {
253             return NullString;
254         }
255         else
256         {
257             string sTmp(sptr + strlen(sptr) - iLen);
258             return sTmp;
259         }
260     }
261
262     ////////////////////////tolower/////////////////////////////////////////
263     void string::tolower()
264     {
265         for (int iChar = 0; iChar < length(); iChar ++)
266         {
267             *(sptr + iChar) = ::tolower(*(sptr + iChar));
268         }
269     }
270
271
272     ////////////////////////toupper/////////////////////////////////////////
273     void string::toupper()
274     {
275         for (int iChar = 0; iChar < length(); iChar ++)
276         {
277             *(sptr + iChar) = ::toupper(*(sptr + iChar));
278         }
279     }
280
281     ////////// substring: middle len chars starting from where ///////////////
282     string string::mid(int iLen, int iWhere) const
283     {
```

```
284         if (iLen <= 0)
285         {
286             return NullString;
287         }
288         if (iWhere < 0 || iWhere >= length())
289         {
290             return NullString;
291         }
292         string sTmp(iLen+1);
293         strncpy(sTmp.sptr, sptr + iWhere, iLen);
294         sTmp[iLen] = '\0';
295         return sTmp;
296     }
297
298     /* the declarations from the file strings.ipp */
299
300     string::~string(void)
301     {
302         destroy(sptr, iSize);
303     }
304
305     bool string::operator==(const string & sOperand) const
306     {
307         return (strcmp(sptr,sOperand.sptr) == 0);
308     }
309
310     bool string::operator!=(const string & sOperand) const
311     {
312         return (strcmp(sptr,sOperand.sptr) != 0);
313     }
314
315     bool string::operator<(const string & sOperand) const
316     {
317         return (strcmp(sptr, sOperand.sptr) <0);
318     }
319
320     bool string::operator>(const string & sOperand) const
321     {
322         return (strcmp(sptr, sOperand.sptr) > 0);
323     }
324
325     bool string::operator<=(const string & sOperand) const
326     {
327         return (strcmp(sptr, sOperand.sptr) <= 0);
328     }
329
330     bool string::operator>=(const string & sOperand) const
331     {
332         return (strcmp(sptr, sOperand.sptr) >= 0);
333     }
334
335     int string::length(void) const
336     {
337         return strlen(sptr);
338     }
339
340     int string::storesize(void) const
341     {
342         return iSize;
343     }
344
345     string::operator const char * () const
346     {
347         return (const char *) sptr;
348     }
349
350     bool string::empty(void) const
351     {
352         return (length() == 0);
353     }
354
355     char string::operator[](int n) const
356     {
357         cCrud = '\0';
358         if (n > iSize || n <= 0)
359         {
360             return cCrud;
361         }
362         else
363         {
```

```
364             return *(sptr + n);
365         }
366     }
367
368     char & string::operator[](int n)
369     {
370         cCrud = '\0';
371         if ((n > iSize) || (iSize <= 0))
372         {
373             return (char &) cCrud;
374         }
375         else
376         {
377             return (char &) *(sptr + n);
378         }
379     }
```

Appendix 4

The Class SimpleLex

The implementation of the class SimpleLex:

```
1   /* This class implements the simple lexical analyse which detects the
2    * simple collection of tokens such as identifiers and numbers
3    * AUTHOR C.D. Turner
4    * DATE 20/8/92
5    */
6
7   #include "simpllex.hpp"
8
9   SimpleLex::SimpleLex(StringOfTokens * tsData) : BasicLex(tsData)
10  {
11      if (tsData == NULL)
12          tkThis = SimpleLex::Error;
13      else
14          tkThis = SimpleLex::Unknown;
15      iStartPos = 0;
16  }
17
18  SimpleLex::~SimpleLex()
19  {
20  }
21
22  SimpleLex::Token SimpleLex::current()
23  {
24      return tkThis;
25  }
26
27  SimpleLex::TokenSet tsWhitespace(3, BasicLex::Space, BasicLex::Tab,
28      BasicLex::ControlChar);
29
30  SimpleLex::Token SimpleLex::next()
31  {
32      if (sotData == NULL)
33          return SimpleLex::Error;
34
35      if (BasicLex::tkThis != BasicLex::EndOfFile)
36      {
37          BasicLex::Token btkCur = BasicLex::next();
38          iStartPos = iCurPos;
39
40          switch(btkCur)
41          {
42              case BasicLex::EndOfFile:
43                  tkThis = SimpleLex::EndOfFile;
```

```
44                  break;
45              case BasicLex::EndOfLine:
46                  tkThis = SimpleLex::EndOfLine;
47                  break;
48              case BasicLex::Symbol:
49                  tkThis = SimpleLex::Symbol;
50                  break;
51              case BasicLex::Tab:
52              case BasicLex::Space:
53              case BasicLex::ControlChar:
54                  do
55                  {
56                      btkCur = BasicLex::next();
57                  }
58                  while (tsWhitespace.hasmember(btkCur) == true);
59                  tkThis = SimpleLex::Whitespace;
60                  iCurPos --;
61                  break;
62              case BasicLex::Digit:
63                  do
64                  {
65                      btkCur = BasicLex::next();
66                  }
67                  while (BasicLex::IsLookAhead(BasicLex::Digit, 0) == true);
68                  tkThis = SimpleLex::Number;
69                  iCurPos --;
70                  break;
71              case BasicLex::Alpha:
72                  do
73                  {
74                      btkCur = BasicLex::next();
75                  }
76                  while (BasicLex::IsLookAhead(BasicLex::Alpha, 0) == true);
77                  tkThis = SimpleLex::Identifier;
78                  iCurPos --;
79                  break;
80          }
81      }
82      return tkThis;
83  }
84
85  void SimpleLex::Text(string & sLine)
86  {
87      if (sotData == NULL)
88          return;
89
90      int iLen = iCurPos - iStartPos + 1;
91      if (iLen >= 0)
92          sLine = sotData->Line().mid(iLen, iStartPos);
93  }
94
95  bool SimpleLex::MoveToNextLine()
96  {
97      iStartPos = 0;
98      bool bRes = BasicLex::MoveToNextLine();
99      if (bRes)
100         tkThis = SimpleLex::Unknown;
101     else
102         tkThis = SimpleLex::EndOfFile;
103     return bRes;
104 }
105
106 SimpleLex::Token SimpleLex::skipto(const SimpleLex::TokenSet & tsSkip)
107 {
108     if (sotData == NULL)
109         return Error;
110
111     SimpleLex::Token tkTmp = this->SimpleLex::current();
112     while ((tkTmp != SimpleLex::EndOfFile) &&
113         (tsSkip.hasmember(tkTmp) == false))
114         tkTmp = next();
115
116     return tkTmp;
117 }
```

Appendix 5

The Class Lex

The implementation of the class Lex:

```
/* this file contains the code for the lexical analyser
 * for the creation of executable code from the test cases
 * AUTHOR C.D.Turner
 * DATE 12/8/92
 * NOTE:
 * altered, it is now derieved from the class simplelex.
 */

#include <ctype.h>
#include <iostream.h>
#include "..\lex\lex.hpp"

// array used to speed up the searching for reserved words
const int caSpeeder[] =
{
    -1, -1,
    0, 4,
    6, 8,
    11, 13,
    14, -1,
    -1, -1,
    16, 17,
    20, 22,
    -1, -1,
    24, 32,
    35, 36,
    -1, -1,
    -1, -1
};

const static struct MatchReserved
{
    char * sReserved;
    lex::token tkReserved;
} mrReserved[] =
{
    "CASE", lex::tkiCASE,
    "CHANGE", lex::tkiCHANGE,
    "CLASS", lex::tkiCLASS,
    "CODE", lex::tkiCODE,
    "DATA", lex::tkiDATA,
    "DATE", lex::tkiDATE,
    "END", lex::tkiEND,
```

```
    "ENGINEER", lex::tkiENGINEER,
    "FEATURE", lex::tkiFEATURE,
    "FILES", lex::tkiFILES,
    "FINISH", lex::tkiFINISH,
    "GENERATOR", lex::tkiGENERATOR,
    "GENERATORS", lex::tkiGENERATORS,
    "HAS", lex::tkiHAS,
    "INCLUDE", lex::tkiINCLUDE,
    "INVARIANT", lex::tkiINVARIANT,
    "MULTIPLE", lex::tkiMULTIPLE,
    "NAME", lex::tkiNAME,
    "NONE", lex::tkiNONE,
    "NUMBER", lex::tkiNUMBER,
    "OBJECT", lex::tkiOBJECT,
    "OF", lex::tkiOF,
    "POSTCONDITION", lex::tkiPOSTCONDITION,
    "PRECONDITION", lex::tkiPRECONDITION,
    "S", lex::tkiS,
    "SCENARIO", lex::tkiSCENARIO,
    "SCENARIOS", lex::tkiSCENARIOS,
    "SINGLE", lex::tkiSINGLE,
    "START", lex::tkiSTART,
    "STATES", lex::tkiSTATES,
    "SUBSTATE", lex::tkiSUBSTATE,
    "SUBSTATES", lex::tkiSUBSTATES,
    "TERMINATOR", lex::tkiTERMINATOR,
    "TEST", lex::tkiTEST,
    "TESTS", lex::tkiTESTS,
    "UNDER", lex::tkiUNDER,
    "VALUES", lex::tkiVALUES
};

const int ciReserved = sizeof(mrReserved) / sizeof(MatchReserved);


static const char * sTokens[] =
{
    ":",
    "=",
    ".",
    "..",
    "{{",
    "}}",
    "[",
    "]",
    "(",
    ")",
    ",",
    "&",
    "|",
    "!",
    "*",
    "?",
    "$",
    "->",
    "-!",
    ";",
    "<number>",
    "\"string\"",
    "<filename>",
    "<identifier>",
    "End Of File",
    "UNKNOWN token",
    "Error:internal token"
};

lex::MatchTokens::MatchTokens(lex::token tkObject, const bool &  bObject)
{
    tkThis = tkObject;
    bStore = bObject;
};

////////////////MatchTokenString()////////////
bool lex::MatchTokenString(lex::MatchTokens mtArray[],
    int iSize, int & iPos)
{
    token tkCur = lex::reserved();

    // use current symbol on first pass, and pick up next() for
    // subsequent passes.
```

```
    bool bFirstPass = true;

    for (iPos = 0; iPos < iSize; iPos ++)
    {
        if (bFirstPass == true)
            bFirstPass = false;
        else
        {
            tkCur = lex::next();
            tkCur = reserved();
        }
        if (tkCur != mtArray[iPos].tkThis)
            return false;
        if (mtArray[iPos].bStore == true)
        {
            string sTmp;
            lex::text(sTmp);
            mtArray[iPos].sData = sTmp;
        }
    }
    return true;
}

////////////////////tokenstring//////////////////////////////////
const char * lex::tokenstring(token tkPrint) const
{
    if ((tkPrint >= tkiCASE) && (tkPrint <= tkiVALUES))
        return mrReserved[tkPrint].sReserved;
    else
        return sTokens[tkPrint - tkiVALUES - 1];
}

////////////////////////////////////lex()/////////////////////////////////////
lex::lex(StringOfTokens * ptsData) : SimpleLex(ptsData)
{
    if (ptsData == NULL)
        tkNow = tkError;
    else
        tkNow = tkUnknown;
}

////////////////////////////////////~lex()/////////////////////////////////////
lex::~lex()
{
}

////////////////////////////////////next()/////////////////////////////////////
lex::token lex::next()
{
    if (tkNow == tkError)
        return tkNow;

    if (tkNow == tkEOF)
        return tkNow;

    SimpleLex::Token tkCur = SimpleLex::next();

    bool bExit = false;
    do
    {
        if (tkCur == SimpleLex::EndOfFile)
        {
            tkNow = tkEOF;
            return tkNow;
        }

        if ((tkCur == SimpleLex::Whitespace) || (tkCur == SimpleLex::EndOfLine))
        {
            tkCur = SimpleLex::next();
        }
        else if (tkCur == SimpleLex::Symbol)
        {
            // check to see if it is a comment #
            char cTmp;
            BasicLex::Text(cTmp);

            // if a comment, skip it and move onto the next line
            if (cTmp == '#')
            {
                SimpleLex::MoveToNextLine();
```

```
            tkCur = SimpleLex::next();
        }
        // otherwise exit the loop
        else
            bExit = true;
    }
    else if (tkCur == SimpleLex::Error)
    {
        tkNow = tkError;
        return tkNow;
    }
    else if (tkCur == SimpleLex::Unknown)
    {
        tkNow = tkUnknown;
        return tkNow;
    }
    // must have found something valid
    else
        bExit = true;
}
while (bExit == false);

// if reached here, the current character must be something
// valid

char cTmp, cTmp2;
BasicLex::Text(cTmp);
BasicLex::LookAhead(1, cTmp2);

if ((cTmp == '_') || (tkCur == Identifier))
{
    // parse an identifier ..
    tkNow = tkIDENTIFIER;

    // remember the start of the token
    int iRealStart = SimpleLex::iStartPos;

    // now keep scanning if the next character is either a letter,
    // a digit, or an underscore
    BasicLex::Token btkLook = BasicLex::LookAhead(1);
    while ((btkLook == BasicLex::Digit) || (btkLook == BasicLex::Alpha)
        || (cTmp2 == '_'))
    {
        tkCur = SimpleLex::next();
        BasicLex::LookAhead(1, cTmp2);
        btkLook = BasicLex::LookAhead(1);
    }

    // restore the start of the token
    SimpleLex::iStartPos = iRealStart;

}
else if (tkCur == SimpleLex::Symbol)
{
    char cTmp, cTmp2;
    BasicLex::Text(cTmp);
    BasicLex::LookAhead(1, cTmp2);

    if (cTmp ==  ':')
        tkNow = tksCOLON;
    else if (cTmp ==  ';')
        tkNow = tksSEMICOLON;
    else if (cTmp == '=')
        tkNow = tksEQUALS;
    else if(cTmp == '.' && cTmp2 == '.')
    {
        tkNow = tksDEADCOLON;
        tkCur = SimpleLex::next();
    }
    else if (cTmp == '.')
        tkNow = tksDOT;
    else if (cTmp =='{' && cTmp2 == '{')
    {
        tkNow = tksDLEFTBRACE;
        tkCur = SimpleLex::next();
    }
    else if (cTmp == '}' && cTmp2 == '}')
    {
        tkNow = tksDRIGHTBRACE;
        tkCur = SimpleLex::next();
```

```
    }
    else if (cTmp == '(')
        tkNow = tksLEFTBRACKET;
    else if (cTmp == ')')
        tkNow = tksRIGHTBRACKET;
    else if (cTmp == '[')
        tkNow = tksLEFTBOX;
    else if (cTmp == ']')
        tkNow = tksRIGHTBOX;
    else if (cTmp == '*')
        tkNow = tksSTAR;
    else if (cTmp == '?')
        tkNow = tksQUESTION;
    else if (cTmp == '$')
        tkNow = tksDOLLAR;
    else if ((cTmp == '-') && (cTmp2 == '>'))
    {
        tkNow = tksARROW;
        tkCur = SimpleLex::next();
    }
    else if ((cTmp == '-') && (cTmp2 == '!'))
    {
        tkNow = tksNOTARROW;
        tkCur = SimpleLex::next();
    }
    else if (cTmp == ',')
        tkNow = tksCOMMA;
    else if (cTmp == '&')
        tkNow = tksAND;
    else if (cTmp == '|')
        tkNow = tksOR;
    else if (cTmp == '!')
        tkNow = tksNOT;
    else if (cTmp == '\"')
    {
        // remember where the real starting position is
        int iRealStart = SimpleLex::iStartPos;

        tkNow = tkSTRING;
        do
        {
            tkCur = SimpleLex::next();
            BasicLex::Text(cTmp);
        }
        while ((tkCur != SimpleLex::EndOfLine) &&
            (cTmp != '\"'));

        // restore the start of the token
        SimpleLex::iStartPos = iRealStart;

        if (cTmp != '\"')
            tkNow = tkError;
    }
    else if (cTmp == '<')
    {
        // remember where the real starting position is
        int iRealStart = SimpleLex::iStartPos;
        tkNow = tkFILENAME;
        do
        {
            tkCur = SimpleLex::next();
            BasicLex::Text(cTmp);
        }
        while ((tkCur != SimpleLex::EndOfLine) &&
            (cTmp != '>'));

        // restore the start of the token
        SimpleLex::iStartPos = iRealStart;

        if (cTmp != '>')
            tkNow = tkError;
    }
    else
        tkNow = tkUnknown;
    return tkNow;
}
else if (tkCur == SimpleLex::Number)
{
    tkNow = tkNUMBER;
    return tkNow;
```

```
    }
    else
        tkNow = tkUnknown;
    return tkNow;
}

////////////////////////skipto/////////////////////////////////////////////
void lex::skipto(token tkSkipto)
{
    token tkCur;
    tkCur = tkNow;
    while (tkCur != tkSkipto && tkCur != tkEOF && tkCur != tkError)
        tkCur = next();
}

////////////////////////text/////////////////////////////////////////////
void lex::text(string & sText) const
{
    SimpleLex::Text(sText);
}

////////////////////////////readnextline/////////////////////////////////
bool lex::readnextline()
{
    tkNow = tkUnknown;
    return SimpleLex::MoveToNextLine();
}

////////////////////////////reserved()///////////////////////////////////
lex::token lex::reserved() const
{
    if (tkNow != tkIDENTIFIER)
        return tkNow;

    bool bFound = false;
    string sTmp;
    text(sTmp);
    int iPos = ((int) sTmp[0]) - int('A');
    if ((iPos < 0) || (iPos > 25))
        return tkIDENTIFIER;

    // speed up the search by starting at the appropriate letter
    int iNumTried = caSpeeder[iPos];
    if (iNumTried == -1)
        return tkIDENTIFIER;

    while (iNumTried < ciReserved)
    {
        int iRes = strcmp(sTmp, mrReserved[iNumTried].sReserved);
        if (iRes == 0)
        {
            bFound = true;
            break;
        }
        else if (iRes < 0)
            break;
        iNumTried ++;
    }
    if (bFound == true)
        return mrReserved[iNumTried].tkReserved;
    else
        return tkIDENTIFIER;
}

////////////////////////////line()//////////////////////////////////////
void lex::line(string & sLine) const
{
    SimpleLex::Line(sLine);
}

////////////////////////////linenum()//////////////////////////////////
int lex::linenum() const
{
    return SimpleLex::LineNum();
}

lex::token lex::current(void) const
{
    return tkNow;
}
```