Testing Graph Classes

by

Kshitij Kumar
B Tech , Maulana Azad College of Technology, Bhopal, India, 1991

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in the
Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. D. M. Hoffman, Supervisor (Dept. of Computer Science)

[redacted]
Dr. G. C. Shoja, Departmental Member (Dept. of Computer Science)

[redacted]
Dr. N. Dimopoulos, Outside Member (Dept. of Electrical & Computer Engineering)

[redacted]
Dr. B. Freeman-Benson, External Examiner (Adjunct, Dept. of Computer Science)

University of Victoria

Supervisor: Dr. D. M. Hoffman

Abstract

Object-oriented languages promote code reuse, through the development of widely available class libraries. Code reuse implies that the classes being reused need to be well tested. We present a graph class used to store and traverse directed *testgraphs* as part of the TestGraph Methodology for testing of C++ classes. A testgraph is a partial model of the state-transition graph of the class being tested. We focus on providing suites for the automated testing of collection classes. We present a scheme to test graph classes, by placing graphs including ring, star, and complete graphs into an object of the graph class. We first apply this scheme to test the `TestGraph` class. We then demonstrate the graph testing scheme in a real-world setting by using it to test a commercially available graph class. Substantial testing for the commercial graph class was achieved at comparatively low cost.

Examiners:

[redacted]

Dr. D. M. Hoffman, Supervisor (Dept. of Computer Science)

[redacted]

Dr. G. C. Shoja, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. N. Dimopoulos, Outside Member (Dept. of Electrical & Computer Engineering)

[redacted]

Dr. B. Freeman-Benson, External Examiner (Adjunct, Dept. of Computer Science)

Table of Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr. D. M. Hoffman of the Department of Computer Science, for his encouragement, patience, and advice during the course of this research and for his help in the preparation of this manuscript. Financial assistance received from Dr. Hoffman under NSERC grant 0GP0008067, and that from the University of Victoria, is also gratefully acknowledged.

To my wife, who was by my side through good times and bad, to my parents, who made me what I am, and to God, for everything

Chapter 1

Introduction

1.1 The Problem

The Computer Science community, both academic and industrial, has been steadily adopting object-oriented software development methodologies over the last few years. object-oriented languages offer certain advantages over traditional languages, such as code reuse, better reliability and well-defined class interfaces, they also offer useful features such as encapsulation, inheritance, and polymorphism. Commercially available class libraries supply building blocks for other applications, providing often-used functionality and hence promoting code reuse.

There is, however, a hidden cost associated with object-oriented software development. The code being reused needs to be unusually well tested, since many applications depend on its correctness. The presence of well-defined class interfaces implies that interfaces must be tested well. The implementer of a class exploiting inheritance needs to be confident that the functionality expected from the parent class is actually provided. Several other issues appear during testing of object-oriented software, which make it different from testing traditional software.

Unfortunately, insufficient attention has been paid to the problems of testing object-oriented software. Traditional methods of testing are still being applied to test object-oriented software, even though these methods are often unsuitable for this purpose. Although efforts are now being made to address this problem, the testing of reusable, component classes has still not been given sufficient attention. There exists a strong need for methodologies and tools to test classes in general, and component classes in particular. These tools need to be automated, so that human interaction during testing is reduced, and the tests are easily repeatable.

With this need in mind, the research team I am part of has been developing the testgraph methodology. This methodology is used to test C++ classes, and includes a tool to aid in testing such classes. Our methodology focuses on testing collection classes, rather than Graphical User Interface (GUI) classes. A collection class implements an abstract data type, such as a set, a list, or a graph, on the other hand, GUI classes provide graphical facilities for user interaction, and may even use collection classes.

The contributions from my work here have been the development of the `TestGraph` and `TestGraphIterator` classes, which form the core of the testgraph methodology, followed by my evolving a strategy to test graph classes. The other major contributions were: demonstrating this strategy by applying it to test the `TestGraph` and `TestGraphIterator` classes, and then modifying the graph testing strategy to test a well used graph class from the LEDA class library.

1.2 Testing C++ graph classes

Graph classes are used to store, manipulate and traverse graphs, by users ranging from graph theoreticians developing graph algorithms, to telecommunications engineers modelling telephone networks.

lends confidence in the testgraph methodology, the `TestGraph` classes, and the graph class test scheme

1.3 Thesis overview

Chapter 2 of this thesis reviews work being done in the area of object-oriented testing in general, and class testing in particular, and places our work in perspective Chapter 3 provides the concepts needed to understand the testgraph methodology, and explains the methodology in detail The `TestGraph` classes, their graph traversal algorithm, and other implementation issues are dealt with in Chapter 4, Chapter 5 presents the graph testing scheme used to test the `TestGraph` classes Chapter 6 explains the LEDA graph class, how the graph testing scheme was modified to test this class, and the failures reported from it The thesis research work is summarized, and the directions that future work may take are explained in Chapter 7

Chapter 2

Related Work

Software testing may be specification based or program based [3]. In specification based testing, test cases are based on module specifications, and in program based testing, they are based on the internal structure of the implementation [10]. The test methodology used by us supports both specification and program based testing.

2.1 Testing object-oriented software

Although object-oriented software development has been adopted by the software community in a big way, its testing has not received much attention. Testing of component classes, in particular, seems to have received little research attention. Perry and Kaiser [20] emphasize that there is a pressing need for research on object-oriented testing. They opine that features such as encapsulation and inheritance compound the testing problem. Retesting of inherited functions is recommended in derived classes, and axioms are presented to characterize adequate testing, but no examples are provided of their use.

Smith and Robson [22], in their work on the problem of validation in object-

oriented programming, raise some of the problems caused by features of object-oriented languages, such as C++ [24]. Problems caused because a class cannot be tested dynamically (only its instantiation is tested), and those caused by use of polymorphism and genericity, are raised in [22], but solutions are not suggested for most problems. Arnold and Fuson [1] share observations and lessons learned from object-oriented projects. Their experience has been that techniques that work in small isolation do not scale up well. They provide tips about testing in the real world, and recommend using tools in areas such as memory leak detection. They, however, often resort to non-automated methods, such as completely manual output testing.

Turner and Robson [25] present suggestions for the use of classes and inheritance from the point of view of validation. They present a classification for features of a class, and test the simplest features first, using them to test other features. They do not address the issue of test automation, and do not provide any examples of testing. The need to build testability into object-oriented systems is stressed by Binder [4] and he says that testability reduces cost as well as increases reliability. A set of testability factors are presented, but the issue of actual testing is not discussed. Test automation is addressed in [21], where an example shows how object models can be re-used for testing, reducing the software life cycle by using automated testing tools. The user of this method, however, needs to perform object modelling, and to use the Object Modelling Tool (OMT).

A survey of test methods for object-oriented software is presented in [18], and general approaches are proposed for class and cluster testing. A test framework and strategies to test object-oriented software are presented with a small example in [23]. This framework is not automated, and test results are manually checked. McGregor and Karson [15] present an approach for integrating testing with development in the software life cycle. Testing is begun early in the development cycle, and is interwoven into the development process. Test suite automation and regression testing are

suggested, but no concrete methods are provided

2.1.1 Unit testing

The IEEE [12] defines unit testing as

> A process that includes the performance of test planning, the acquisition of a test set, and the measurement of a test unit against its requirements

There seems to be no general agreement on the unit for object-oriented testing In [13], where object-oriented constructs for integration testing are presented, object methods are treated as units, and method testing is referred to as unit testing The authors give a small example of testing methods as units using stubs, but have not used their method to test large software Fiedler [5] describes an approach to unit testing in object-oriented programs, with test strategies for each stage from test design to output verification He treats a class as the unit of testing Once again, human verification of output results is required

Harrold, et al [14], present a strategy for testing classes that inherit from other classes They provide a strategy where, if base classes are tested first, it can be decided which methods in a derived class need retesting They do not, however, present a test suite Goel and Bashir, [2], test classes according to data, dividing classes into "slices," each slice having one data member and methods acting on it Each slice is tested separately Once again, no test suite is provided as an example of the application of this technique

Class testing tools

ACE, a tool for automated testing of Eiffel and C++ classes, is presented by Murphy et al [6], where the testing earlier involved only "cluster" testing (testing of several related classes together) Improved testing is reported since it was modified to use

class testing and the tool, but the authors foresee problems in testing in larger frameworks. ACE is based on an earlier tool for testing C modules [7]. A suite of tools is presented in [26], where a tool generates a set of test cases upon input of a test script. The tester has to learn the formalisms used to write the scripts, which specify states, function pre and postconditions, and invariants on the class under test, used to create test cases. No examples of its use are provided, since only a prototype has been developed.

The testgraph tool and methodology have been explained in [11], and are adapted to test a commercial class library in [8]. Exploiting inheritance in the classes under test by replicating the hierarchical structure in the test suite is illustrated with examples in [9].

Chapter 3

Terms and Concepts

3.1 Class Interface Specification

A *Class Interface Specification* (CIS) describes the observable behaviour of a C++ class. The CIS reflects the class behaviour as specified by the class designer, and which an implementer tries to provide in the class implementation. A CIS documents the set of assumptions that users of the class can make about it, and which the implementers of the class make about its use. Users and testers of the class read the CIS to determine the expected class behaviour.

The CIS is contained in a C++ *header file*. A header file is a C++ source file with a name ending in "`.h`". For example, `IntSet.h` is the header file for the `IntSet` class. The CIS has two parts: the syntax and the semantics.

3.1.1 Interface Syntax

The *interface syntax* declares the types and constants exported by the class, and the names, parameters, and return value types of its member functions and exception handlers.

The **types** and **constants** sections declare exported types and constants. The **exception handlers** section specifies the function prototypes for the exception handlers for the class.

The **class declaration** section declares the class name, public member functions in the class, their parameter and return value types, and public instance variables. In addition, private members of the class are also declared here. These are marked as "private information" since they need to be specified here following the C++ syntax for a class declaration, but are not available to the class user.

The **concrete state invariant** and **abstraction function** sections also constitute private information. They are included here since the concrete state, specified by instance variables, is specified here. The concrete state invariant is a predicate on the concrete state of the class, designed to restrict legal states. If the invariant holds when a routine is invoked, it should hold when program execution returns from the routine. The abstraction function gives a mapping from the legal concrete states of the class to its abstract states.

Figure 3.1 shows the interface syntax for the `IntSet` class which stores a bounded set of integers. The public member functions `add` and `remove` are used to add or remove elements from the set, `isMember` checks for element membership, and `size` provides the set size. The concrete state is specified by private variables - `s`, an array of integers to store the set, and `curSize`, an integer to store the current set size. The class also has a private member function, `findPos`.

3.1.2 Interface Semantics

The *interface semantics* section specifies the abstract state of the class and describes the behaviour of each exported member function of the class in terms of its abstract state. The **abstract state variables** section specifies the variables used to represent

```
// *****interface syntax*****

// ***constants***
const int MAXSIZE = 100;

// ***types***

// ***class declaration***
class IntSet {
public
        IntSet(),

        // set calls
        void add(int),
        void remove(int),

        // get calls
        int isMember(int) const,
        int size() const,

// ***private information---begin***
private
        int findPos(int) const,
        int s[MAXSIZE],
        int curSize,
};

// ***concrete state invariant***
//      1. There are no duplicates in s[0..curSize-1]
//      2. curSize <= MAXSIZE
//
// ***abstraction function***
//      s = {x | (exists i in [0..curSize-1])(x == s[i])}
// ***private information---end***

// ***exception handlers***
void duplicateExc(),
void fullExc(),
void notFoundExc(),
```

Figure 3.1 `IntSet` Class Interface Specification—interface syntax

the abstract state The **abstract state invariant** section specifies a predicate on the abstract state space, that restricts the meaningful states of the class After every access routine call, the abstract state should satisfy the abstract state invariant

The **member function semantics** section contains one subsection for each public member function in the class, specifying its normal case and exceptional behaviour, and assumptions for the function

Functions calls may be *set*, *get*, or *set-get* Set calls change the internal state of the class, get calls return values based on the internal state of the class, and set-get calls do both

The normal case behaviour of the function is represented by a **transition** entry for each set call, an **output** entry for each get call, and a **transition-output** entry for each set-get call The **transition** entry specifies a state transition new values for the state variables expressed in terms of the old values and the function parameters The **output** entry specifies the return value from the function by an assignment to the special variable `out` The **transition-output** describes both a transition and an output, the transition specified as in a set call and the output specified as in a get call Normal case behaviour occurs only if no exception is signalled

The **exceptions** entry in each subsection specifies the illegal conditions which should be handled by signalling an exception Exceptions are specified by assignment to the special variable `exc`, or "none" if no exception is signalled If an exception occurs, no change is made to the object state

The assumptions for a function express those illegal conditions which the function need not handle A function produces the required behavior if the assumption for the function holds at the time of the function invocation The **assumptions** entry in each subsection specifies the assumptions for the function, or "none" if there are no assumptions

Figure 3 2 shows the interface semantics for `IntSet` The abstract state is repre-

```
***abstract state variables***
s: set of integer

***abstract state invariant***
|s| <= MAXSIZE

***member functions***

IntSet:
        transition: s = {}
        exceptions: none
        assumptions: none

add(x):
        transition: s = s + {x}
        exceptions: exc = (x in s => duplicateExc
                           | |s| == MAXSIZE => fullExc)
        assumptions: none

remove(x):
        transition: s = s - {x}
        exceptions: exc = (x not in s => notFoundExc)
        assumptions: none

isMember(x):
        output: out = (x in s)
        exceptions: none
        assumptions: none

size():
        output: out = |s|
        exceptions: none
        assumptions: none
```

Figure 3.2 `IntSet` Class Interface Specification—interface semantics

sented by a set of integers, `s`. The abstract state invariant permits the size of the set to be at most `MAXSIZE`.

The normal case member function semantics are straightforward. The constructor initializes the `IntSet` instance to the empty set. The function `add(`x`)` adds x to the set, and `remove(`x`)` deletes the element x from the set. `isMember(`x`)` returns true or false depending on whether x is in the set and `size` returns the size of the set.

The function `add(`x`)` signals the exception `duplicateExc` if x is a member of the set, and `fullExc` if the set is full. Exception `notFoundExc` is signalled from `remove(`x`)` if x is not a member of the set.

3.1.3 Exception Signalling

A CIS specifies when exceptions are raised, but does not state how exception signalling must be done. Our exception handlers are implemented as function calls with the same name as the exceptions. Exceptions could have been signalled using the C++ language constructs `throw`, `catch`, and `try`. However, most C++ compilers available today do not provide these features, and so we did not use them.

Figure 3.3 shows the C++ implementation for `IntSet::add`. It is in terms of the `IntSet` concrete state and closely resembles the member function semantics of `add` from Figure 3.2. The exception conditions are checked first, and the concrete state transition is made if no exception occurs. If `findPos(x)` returns a valid position in `s`, then `x` already exists in the set, and `duplicateExc` is raised. Otherwise, if the set already consists of `MAXSIZE` number of elements, `fullExc` is raised. If not, the transition is made on the concrete state.

```
void IntSet::add(int x)
{
        if (findPos(x) >= 0) {
                duplicateExc();
                return;
        } else if (curSize == MAXSIZE) {
                fullExc();
                return;
        }
        s[curSize++] = x;
}
```

Figure 3.3: `IntSet` class—`add` implementation

3.2 The testgraph methodology

3.2.1 The Test Programmer's tasks

The Test programmer uses the *Testgraph Editor* (TGE), to create a *testgraph*. A testgraph is a graph representing an abstraction of the state-transition graph of the *Class Under Test* (CUT).

The `Oracle` class is developed to function as the *test oracle*. It is used for test output checking, and also for test input generation.

The `Driver` class is developed as part of the *test driver*, which is used for loading and traversing the testgraph, and for verifying the CUT behaviour.

3.2.2 Testgraphs

Testgraphs are directed graphs having unique integer node labels and integer arc labels. Testgraphs have a designated *start node*.

A testgraph is a 3-tuple $\langle N, A, s \rangle$ where N is a set of integers, A is a set of arcs, and s is an integer. An arc is a tuple of $\langle src, dst, label$ integer$\rangle$ such that $src, dst \in N$.

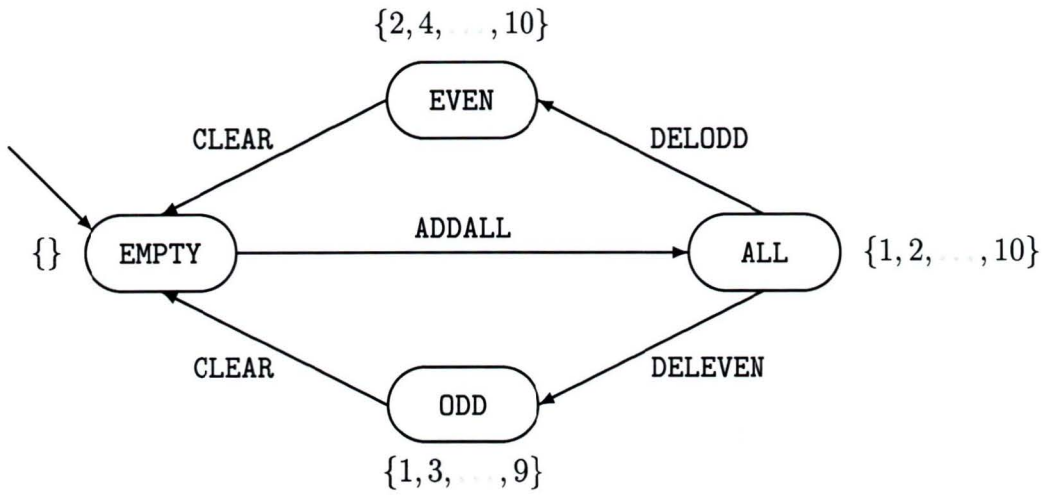


Figure 3 4 Testgraph for **IntSet** (test suite parameter = 10)

A *path* in a testgraph is a sequence $\langle n_0, n_1, \ldots, n_r \rangle$ where for every $i \in [0, r]$, $n_i \in N$ and for every $j \in [1, r]$, it contains an arc from n_{j-1} to n_j A *rooted path* starts at the start node

A testgraph is an abstraction a testgraph node corresponds to one or more CUT state-transition graph nodes, and a testgraph arc corresponds to one or more CUT state-transition graph arcs Testgraph nodes represent states the CUT assumes during the testing process Arcs between nodes depict transitions between these CUT states Rooted paths represent a sequence of CUT testgraph states and transitions, starting with the state represented by the start node

A *test suite parameter* is associated with each test suite Varying the test suite parameter values allows us to change the portion of the CUT state graph to be traversed during testing

Figure 3 4 shows the testgraph for **IntSet**, with the state represented by each node shown beside it The test suite parameter is the maximum size of the set, 10 The node which is the destination of an arc with no source node is the start node, **EMPTY**,

representing the empty set. The arc `ADDALL` depicts the transition from the empty set to a full set represented by the node `ALL`. This transition is achieved by adding all integers from 1 to 10 to the empty `IntSet`. Similarly, the other arcs represent the transitions from the states represented by their source nodes to those represented by their destination nodes. A set of rooted paths covering all arcs in the testgraph in Figure 3.4 is:

⟨`EMPTY`, `ALL`, `EVEN`, `EMPTY`⟩

⟨`EMPTY`, `ALL`, `ODD`, `EMPTY`⟩

A testgraph could conceivably have multiple arcs between a pair of nodes. Each arc between two testgraph nodes would correspond to a different way of making the transition between their corresponding CUT states. Such graphs, called multigraphs, are not supported in our current testgraph scheme.

3.2.3 Oracle

Our test oracle is the `Oracle` class, which functions as an automated source of authoritative answers to queries regarding the expected CUT state. The `Oracle` class is like the CUT, but handles only the states and transitions in the CUT testgraph. `Oracle` member functions provide information about the expected CUT state at each testgraph node. The `Oracle` class is also a source of input values for changing the CUT state.

Figure 3.5 shows the `Oracle` class declaration for `IntSet`. `Oracle` member functions intentionally resemble `IntSet` functions. The `Oracle` functions `add` and `remove` are used to modify the `Oracle` to represent the `IntSet` at different testgraph nodes. The function `isMember` is used to test for element membership in the current `IntSet`.

Instance variables `node` and `maxSize` together represent the concrete state of the

```
class Oracle {
public:
        Oracle(int,int=EMPTY);

        void add(Oracle&);
        void remove(Oracle&);

        int isMember(int) const;
        int size() const;

        int node; // current testgraph node identifier
        int maxSize; // current test suite parameter
};
```

Figure 3.5 `IntSet Oracle` class—interface syntax

`Oracle maxSize` maintains the current value of the test suite parameter, and `node` stores the current testgraph node. The ⟨`node`,`maxSize`⟩ pair represents the set composed of integers selected from `[1 maxSize]`, according to the `IntSet` state associated with `node`. The integer `node` is represented as a two-bit binary number with the left bit standing for the even elements and the right bit for the odd ones in the `IntSet` instance, as follows:

`EMPTY` 00 `ODD` 01 `EVEN` 10 `ALL` 11

For example, in Figure 3.4, the pair ⟨`ALL`, 10⟩ represents {1,2, ...,10}.

3.2.4 Driver

The test suite driver runs the system used for regression testing. It collects the actual outputs and compares them to the expected outputs. As part of the driver, our `Driver` class produces the transitions associated with testgraph arcs, and checks the

```
class Driver : public AbstractDriver {
public:
        Driver(int);
        void reset();
        void arc(int);
        void node();

private:
        IntSet cut;
        Oracle orc;

        void loadCut(IntSet&,Oracle&);
        void deleteCut(IntSet&,Oracle&);
        void checkCut(IntSet&,Oracle&);

        void checkDuplicateExc(IntSet&,Oracle&);
        void checkFullExc(IntSet&,Oracle&);
        void checkNotFoundExc(IntSet&,Oracle&);
};
```

Figure 3.6 `IntSet Driver` class—interface syntax

CUT state associated with testgraph nodes.

The `Driver` contains private instances of the CUT and `Oracle`. When a testgraph arc is traversed, the `Driver` member function `arc` is invoked, which produces the transition associated with the arc on both the CUT and the `Oracle`. When a testgraph node is visited, the `Driver` function `node` is invoked, which tests the actual CUT state against the expected state provided by the `Oracle`.

The Driver class derives from an abstract base class, `AbstractDriver`, which provides declarations for all those functions which appear in every `Driver` class.

Figure 3.6 shows the `Driver` class for `IntSet`. The instance variable `cut` is the instance of `IntSet` to be tested, and `orc` is the instance of `Oracle` to test the CUT against.

When an arc in the testgraph from Figure 3.4 is traversed, `arc` invokes `Oracle`

functions to modify `orc` to reflect the CUT state associated with the destination node `arc` also invokes the functions `loadCut` and `deleteCut` to produce the arc transition on `cut`

When traversing a node, `node` invokes `checkCut` to check the normal case behaviour and uses private member functions `checkDuplicateExc`, `checkFullExc`, and `checkNotFoundExc` to check the exceptional behaviour of the CUT

3.2.5 Test Plan

The *Test Plan* (TP) is intended for the testers and maintainers of a class, and serves as a specification of the test implementation for the CUT It is used to plan, document, and maintain the testing for the CUT Planning the testing involves deciding on how to test the normal case and exceptional behaviour, how much code coverage to achieve, and selecting the test cases which would best exercise the CUT

Once the test suite has been implemented, the strategy used for selecting and executing the test cases is documented in the TP, to facilitate test maintenance

TP sections

The values for the test suite parameter are provided in a TP section entitled **Test Suite Parameters** The **Assumptions** section states any assumptions made by the test programmer, and not contained in the CIS **Test Case Selection Strategy** outlines the criteria used for selecting the test cases The **Testgraph** section provides a tabular representation of the testgraph used for testing the CUT **Oracle Strategy** and **Driver Strategy** explain the design of the `Oracle` and `Driver`

For instance, Figure 3 7 shows part of the test plan for `IntSet` The test suite parameter, `maxSize`, is an integer representing the maximum number of elements to be stored in the `IntSet` instance during testing Interesting values for `maxSize`

```
TEST SUITE PARAMETERS
        maxSize: integer
        Selected values: {2,5,10,100}

TEST CASE SELECTION STRATEGY
Specification-based tests
        Normal case
            Special values for object state
                Interval rule on set size
        ...

        Exceptions
            Generate each exception at least once
        ...

ORACLE STRATEGY
        Instance variables
            Represent the nodes using a 2-bit string:
                EMPTY: 00 ODD: 01 EVEN: 10 ALL: 11

        Public member functions
            add: use C bit operators
            remove: use C bit operators
        ...

DRIVER STRATEGY
        Instance variables
            cut and orc to store current CUT and oracle

        Public member functions
            arc
                Four-way case statement using loadCUT, delCUT
        ...

        Private member functions
            loadCUT(IntSet& cut1,Oracle& orc1)
                add every element in orc1 to cut1

        ...
```

Figure 3.7 `IntSet` class—Test Plan

include `MAXSIZE`, to represent the boundary condition. The sizes of the sets to be tested are obtained by applying the interval rule on set size. Sets of size 0, `MAXSIZE`, and some values in between are tested.

3.3 Building and traversing testgraphs

The TGE displays testgraphs in a tabular form, as a square matrix with nodes on the main diagonal and arcs elsewhere. The start node always occupies the first entry in the first row. For an arc, the source node label, the arc label, and the destination node label are placed in clockwise order on the graph. A variety of menu options are provided to modify the testgraph and to store it on disk.

Figure 3.8 shows the TGE with the `IntSet` testgraph. As expected, the node occupying the first entry in the top row is the start node, `EMPTY`. `EMPTY` is the source node for the arc `ADDALL` and its destination node is `ALL`. The node `ALL` is the source for the arc `DELODD`, and its destination is the node `EVEN`, and so on.

The testgraph stored on disk is loaded by the test driver and is stored in an instance of the `TestGraph` class. The driver traverses rooted paths in the testgraph, generated using the `TestGraphIterator` class. `Driver` functions `arc` and `node` are invoked by the test driver whenever an arc or node in the testgraph is traversed.

The algorithm used by the `TestGraphIterator` class provides arc (and node) coverage of the testgraph, given that all arcs (and nodes) are reachable from the start node.

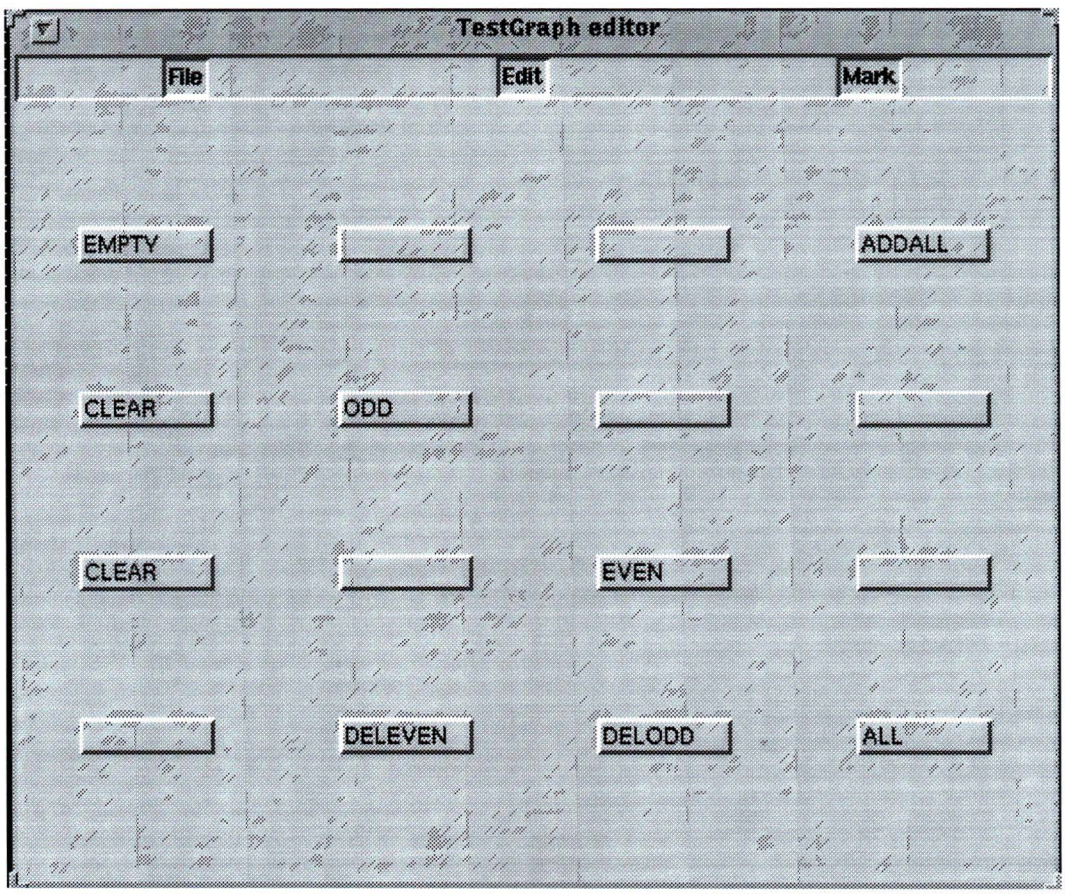


Figure 3.8 The TestGraph Editor—user interface

Chapter 4

The TestGraph Classes

The tester using the testgraph methodology needs to store, modify, and traverse testgraphs. The `TestGraph` and `TestGraphIterator` classes provide this functionality, the former is used to build testgraphs and the latter to traverse paths through them.

4.1 The `TestGraph` class

The `TestGraph` class provides the means to create, store and modify a testgraph. Member functions of the class include set calls used to add or delete arcs and nodes in the testgraph, and get calls used to check for arc and node existence.

4.1.1 `TestGraph` Interface Specification

Figure 4.1 shows part of the interface syntax for the `TestGraph` class, and Figure 4.2 shows part of the interface semantics for it. The complete CIS is given in Appendix A. The abstract variables `nodeSet` and `arcSet` (Figure 4.2) represent the sets of nodes and arcs in the testgraph, and `startNode` represents the testgraph start node. The abstract state invariant excludes multigraphs from valid states. The size of `nodeSet`

```
class TestGraph {
public:
        friend class TestGraphIterator;

        TestGraph(int start = 0);

        void setStartNode(int); // node calls
        int getStartNode() const;
        void addNode(int);
        void delNode(int);
        int exNode(int) const;
        int getNumNode() const;

        void addArc(int,int,int); // arc calls
        void delArc(int,int);
        int exArc(int,int) const;
        int getLabel(int,int) const;

...
};
// ***exception handlers***
void TGExArcExc();
void TGExNodeExc();
void TGFullExc();
void TGNotExArcExc();
void TGNotExNodeExc();
```

Figure 4.1 `TestGraph`—interface syntax

is at most `MAXNODES`, and the source and destination for each arc in `arcSet` have to be nodes from `nodeSet`.

Set calls

The call `addNode(`x`)` adds a new node, x, to `nodeSet`, signalling `fullExc` if `nodeSet` is full, and `TGExNodeExc` if x already exists in `nodeSet`. The call `delNode(`x`)` deletes node x from `nodeSet`, signaling exception `TGNotExNodeExc` if x is not in `nodeSet`. All arcs for which x is the source or destination node are also deleted. `addArc(`s`,`d`,`l`)`

```
/*****interface semantics---begin*****

***abstract state variables***
        nodeSet  set of integer
        arcSet  set of tuple of <src,dst,label integer>
        startNode  integer

***abstract state invariant***
        |nodeSet| <= MAXNODES and
        (forall arc in arcSet)
                (arc src in nodeSet and arc dst in nodeSet) and
        not (exist arc1,arc2 in arcSet)
                (arc1 src = arc2 src and arc1 dst = arc2 dst)

*****interface semantics---end*****/
```

Figure 4.2: `TestGraph`—interface semantics

adds a new arc with label l to `arcSet`, signalling exception `TGNotExNodeExc` if the source node s or destination node d are not in `nodeSet`, and signalling exception `TGExArcExc` if there already is an arc from s to d in `arcSet`. `delArc(`s`,`d`)` deletes the arc from s to d.

Get calls

The function `getStartNode` returns the identity of `startNode`. `exNode(`i`)` returns true if node i is in the testgraph. If there exists an arc from node s to node d in the testgraph, `exArc(`s`,`d`)` returns true. The function `getNumNode` returns the number of nodes in the testgraph, and `getLabel(`s`,`d`)` returns the label of the arc from s to d in the testgraph.

Setting or deleting the start node

`startNode` is specified when the `TestGraph` object is created, by passing it in as a `TestGraph` constructor parameter, and can be modified later using `setStartNode`. If not passed in as a constructor parameter, `startNode` is set to 0. If the node represented by `startNode` is deleted from `nodeSet` using `delNode`, `startNode` itself remains unchanged. `getStartNode` continues to return the identity of the deleted node as the start node, until `startNode` is explicitly modified. `exNode` can be used to confirm existence of the node obtained using `getStartNode`.

4.1.2 The TestGraph implementation

Storing the TestGraph

Figure 4.3 shows the instance variables from the `TestGraph` class, used to store the testgraph. The integer array `node` stores up to `MAXNODES` nodes. The number of nodes in the testgraph is maintained as `numNode`, and `startNode` maintains the index of the testgraph start node in `node`. The concrete state invariant in Figure 4.4 establishes that `node` holds a set of `numNode` (up to `MAXNODES`) elements. Arcs are stored in `arc`, a two dimensional array of `ArcType`. The maximum number of arcs the class can store is `MAXNODES` $\times$ `MAXNODES`.

Figure 4.3 also shows the definition for `ArcType`. The `label` field holds an arc label, and the `exist` field is true or false, depending on whether or not the `ArcType` entry represents an arc existing in the testgraph. The `exist` and `label` flags for an arc from a node i stored in the `node` array at position i', to a node j stored in `node` at position j', are stored in `arc` at `arc[`i'`][`j'`]`, as shown in Figure 4.5. If `arc[`i'`][`j'`] exist` is set, then there exists an arc from the node at `node[`i'`]` to the one at `node[`j'`]`, with label `arc[`i'`][`j'`] label`.

```
typedef struct ArcType {
        int label,
        int exist,
},
...

int node[MAXNODES], // nodes
ArcType arc[MAXNODES][MAXNODES], // arcs
int numNode, // number of nodes
int startNode, // index of start node
```

Figure 4.3 TestGraph—graph storage

```
// ***concrete state invariant***
//      (numNode in [0 MAXNODES]) and
//      there are no duplicates in node[0 numNode-1]

// ***abstraction function***
//      (|nodeSet| = numNode) and
//       nodeSet = {n | n in node[0 numNode-1]} and
//       arcSet = {<src,dst,label> | (exist i,j in [0 numNode-1])
//                                      (src = node[i] and dst = node[j]
//                                      and label = arc[i][j] label
//                                      and arc[i][j] exist)}
```

Figure 4.4 TestGraph—concrete state invariant and abstraction function

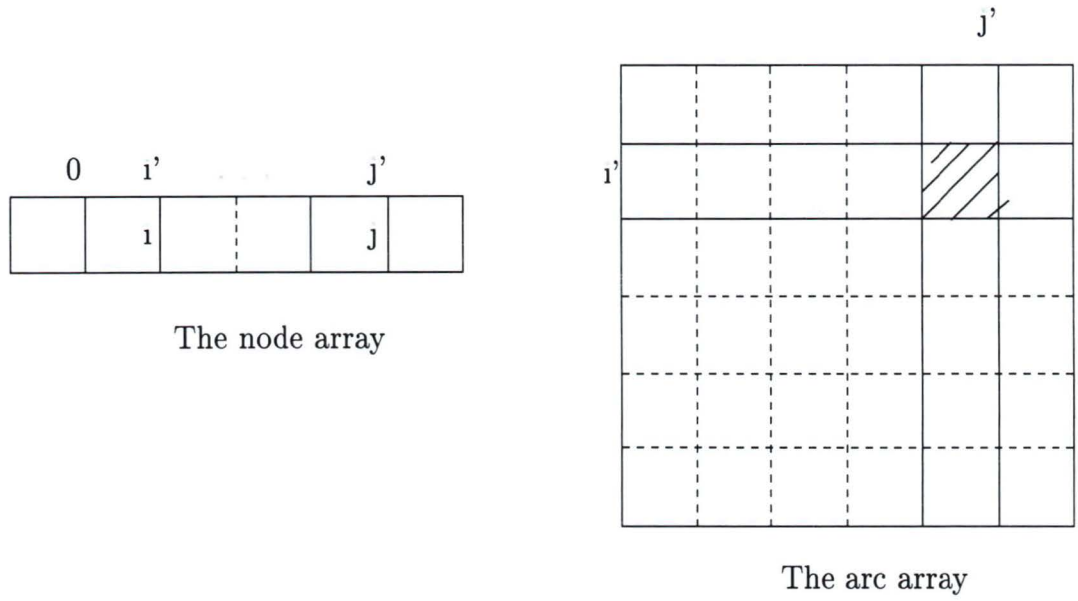


Figure 4.5 The testgraph storage arrays

A testgraph stored in the **TestGraph** class may have at most one arc from a specific source node to a specific destination node in the testgraph. The class can also store self loops, which occur when the destination node of an arc is the same as its source node. The class stores at most one self loop per node in the testgraph.

The abstraction function (Figure 4.4) shows that the concrete state corresponding to **nodeSet** is the **node** array, with **numNode** representing the size of the set. The concrete state corresponding to **arcSet** is **arc**, the source and destination of each arc in the set is a valid **node** element, and each arc is a valid element in the **arc** array.

TestGraph member function implementation

The implementations for **TestGraph** class member functions follow from the member function semantics. For example, Figure 4.6 shows the implementation of **TestGraph addNode**, which adds a node to the testgraph stored in a **TestGraph** object. The exception **TGFullExc** is signaled if **nodeSet** is full. **TGExNodeExc** is signaled if the

```
void TestGraph::addNode(int addNode)
{
        if (numNode == MAXNODES) {
                TGFullExc();
                return;
        }
        if (findPos(addNode) != numNode) {
                TGExNodeExc();
                return;
        }
        // add node
        node[numNode++] = addNode;
        // set arcs to non-existent
        for (int i = 0; i < numNode; i++) {
                arc[i][numNode-1].exist = 0;
                arc[numNode-1][i].exist = 0;
        }
}
```

Figure 4.6 `TestGraph`—`addNode` implementation

node already exists in the testgraph, that is, if the private function `findPos` locates the node in the `node` array. Otherwise, the node is added to the `node` array, and it is ensured that there are no arcs in `arc` for which x can be the source or destination node.

4.2 The `TestGraphIterator`

The class testing tool needs to traverse testgraphs, achieving at least arc coverage. That is, all arcs and nodes reachable from the start node need to be visited. Rooted paths through the testgraph are required, one path at a time.

The `TestGraphIterator` class provides the means to iterate over paths in the testgraph stored in a `TestGraph` object.

```
typedef List<ArcNode> Path;

class TestGraphIterator {
public:
        TestGraphIterator(const TestGraph&);
        ~TestGraphIterator();

        void reset();
        int isEnd();
        Path next();

};

// ***exception handlers***
void TGEndExc();
void TGNoStartExc();
```

Figure 4.7: `TestGraphIterator`—interface syntax

4.2.1 `TestGraphIterator` Interface Specification

The `TestGraphIterator` class has functions to reset the iterator, to check if more paths are left, and to provide the next path. Figure 4.7 shows part of the interface syntax for the `TestGraphIterator` class. A reference to a `TestGraph` object is passed as a constructor parameter. The `TestGraphIterator` class traverses and stores paths from the testgraph in the object referred to by this reference. The paths are traversed using the function `reset`. The function `next` returns the next path in the testgraph. The class provides `isEnd` to check if all paths in the testgraph have been returned to the user.

As shown in Figure 4.8, the abstract state of the `TestGraphIterator` class consists of `graph`, `anSet` and `pSet`. `graph` represents the testgraph, `anSet` the set of arcs and nodes in `graph`, and `pSet` the set of paths in `graph` generated according to the graph coverage criteria. The current graph coverage criteria requires a set of rooted paths,

```
***abstract state variables***
        graph: testgraph
        anSet: set of integer
        pSet: set of path

***abstract state invariant***
        (anSet = graph.arcSet+graph.nodeSet) and
        (forall p in pSet)(a in p => a in anSet)

***graph coverage criteria***
        Every arc reachable from the start node is visited at least once

***iterator policy***
        The iterator will generate a set of rooted paths guaranteed to
        satisfy the coverage criteria

***member functions***
TestGraphIterator(tg):
        transition: graph,anSet,pSet = tg,set of arcs and nodes in tg,
                    set of paths in tg generated according to coverage
                    criteria
        exceptions: exc = (not tg.startNode in tg.anSet => TGNoStartExc)
        assumptions: none

reset():
        transition: pSet = set of paths in graph satisfying the
                    coverage criteria
        exceptions: exc = (not graph.startNode in anSet => TGNoStartExc)
        assumptions: none

next():
        transition-output: pSet,out = pSet-{p},p where p is in pre(pSet)
        exceptions: exc = (pSet == {} => TGEndExc)
        assumptions: testgraph is not modified since last reset call

isEnd():
        output: out = (no unreturned paths in pSet)
        exception: exc = none
        assumptions: none

***local types***
        path: sequence of integer
```

Figure 4.8: `TestGraphIterator`—interface semantics

providing arc coverage

Paths in the `TestGraph` instance are recalculated each time `reset` is invoked If the start node in the testgraph object is invalid, exception `TGNoStartExc` is signaled When a new path is required, the user invokes `next`, which extracts paths from `pSet` If no more paths remain, `next` signals `TGEndExc`

The `next` function behavior is based on the assumption that there have been no changes to the testgraph being traversed since the last time paths in it were calculated The testgraph may be modified later, but these changes are not reflected in the paths returned by `next` until the next time `reset` is invoked The paths returned are the ones in the `TestGraph` object at the time of the last constructor or `reset` invocation

4.2.2 `TestGraphIterator` implementation

Figure 4 9 shows the concrete state for the `TestGraphIterator` class The `testgraph` variable stores a `const` reference to a `TestGraph` object, and provides traversal of paths in that object The two dimensional array of integers, `visited`, is used to mark arcs and nodes visited during the latest testgraph traversal, as shown by the concrete state invariant The path currently being traversed is stored as `curPath`, which is a linked list of `ArcNodes` `ArcNode` is a local class, used for storing arcs and nodes An index, `pathIndex`, maintains the position in `curPath` of the node or arc being currently visited The set of paths generated from the testgraph is stored in `pathSet`, and the iterator, `pathIterator`, iterates over it

The `TestGraphIterator` constructor invokes `reset` to generate paths for the first time `reset` clears `curPath` and `pathSet`, and performs a traversal over `testgraph` by invoking `nextArc` `nextArc` recursively traverses the testgraph and stores the paths generated in `pathSet` The function `next` uses `pathIterator` to iterate over the paths in `pathSet`, and returns a path at a time to the user

```
        const TestGraph& testgraph,
        Path *curPath,
        int first,
        int visited[MAXNODES][MAXNODES], // arcs visited since reset

        int pathIndex, // index of the current node in curPath
        ListSet<Path> *pathSet, // set of paths traversed
        ListSetIter<Path> *pathIterator,

        // local functions
        ListSet<ArcInfo>& getOutArcs(int),
        void nextArc(int),

// ***concrete state invariant***
//     (forall i,j in [0 testgraph numNode-1])(visited[i][j] = 1 =>
//        the arc from testgraph node[i] to testgraph node[j] has been
//        traversed since the iterator was last initialized)
```

Figure 4.9: `TestGraphIterator`—concrete state

4.2.3 Accessing the TestGraph

The `TestGraphIterator` class uses the `friend` construct of C++ to access `TestGraph` class internals. In the `TestGraph` class declaration, (Figure 4.1), the `TestGraphIterator` class is declared as a `friend`, allowing it to access the `TestGraph` private members. The `TestGraph` class instance is accessed using the variable `testgraph`, (declared a `const` reference to `TestGraph`). Making the `testgraph` reference a `const` ensures that it cannot be modified by the `TestGraphIterator` class. The `testgraph` reference is initialized in the `TestGraphIterator` constructor to refer to the `TestGraph` object passed in as a parameter.

4.2.4 Multiple Iterators on one TestGraph

It may be useful to allow multiple iterators to iterate over paths in one TestGraph object at the same time. This would allow the tester to test multiple versions of the CUT with the same testgraph at one time, for example while isolating a bug introduced in a new version of the CUT.

The TestGraphIterator class allows multiple iterators to traverse the same testgraph at a time because multiple TestGraphIterator instances can have references to the same TestGraph class object. Since testgraph is a const reference, changes cannot be made to the TestGraph object by any of the TestGraphIterator instances. Each TestGraphIterator instance also has its own private version of the visited array, allowing the iterators to execute independently. The paths traversed by one instance of TestGraphIterator are unaffected by the paths being traversed by other instances of the class.

4.2.5 Testgraph Traversal Algorithm

The TestGraphIterator class provides testgraph traversal using the local function nextArc, shown in Figure 4.10. Traversal begins at the start node and provides arc coverage. The function nextArc is recursive, and is called from reset. reset creates a new current path (curPath), places the start node at the beginning of the path, and invokes nextArc, passing the start node as parameter. The nextArc(n) call traverses an as yet unvisited arc from the set of out arcs of n, generated by the getOutArcs(n) call. The arc being traversed, and its destination node d, are concatenated to the current path, and nextArc(d) is called recursively.

When the current node has no unvisited out arcs, the end of the current path has been reached. curPath is added to the set of traversed paths, pathSet, and traversal continues from the previous recursive invocation of nextArc. Traversal ends when

```
void TestGraphIterator::nextArc(int curNode)
{
        ListSet<ArcInfo>& outArcSet = getOutArcs(curNode);
        ListSetIter<ArcInfo> iterator(outArcSet);

        while (!iterator.isEnd()) {
                ArcInfo arc = iterator.next();
                if (!visited[arc.src][arc.dst]) {
                        visited[arc.src][arc.dst] = 1;
                        ArcNode arcNode;
                        arcNode.flag = ARC;
                        arcNode.id = arc.label; // arc id
                        curPath->addAt(arcNode,pathIndex++);
                        arcNode.flag = NODE;
                        arcNode.id = testgraph.node[arc.dst]; // node id
                        curPath->addAt(arcNode,pathIndex++);
                        nextArc(arc.dst);
                }
        }

        if (outArcSet.size() == 0) // end of path reached
                pathSet->add(*curPath);

        if ((curNode == testgraph.findPos(testgraph.startNode)) &&
            (curPath->size() == 1)) // if only startnode is in curPath
                curPath->removeAt(--pathIndex); // remove one element
        else { // if more than one elements in outArcSet
                curPath->removeAt(--pathIndex); // remove last node
                curPath->removeAt(--pathIndex); // remove last arc
        }

        delete &outArcSet;
}
```

Figure 4.10: `TestGraphIterator`—graph traveral function `nextArc`

there are no more unvisited arcs reachable from the start node. The complete set of paths is generated and stored until either the object is deleted, or **reset** is called again. Each time **next** is called, an as yet unreturned path from the precalculated set of paths is returned, if such a path exists. Whenever **reset** is invoked, the complete set of paths is recalculated.

It is possible that there are arcs in the graph that are not reachable from the start node. In that case, there is no rooted path which can traverse these arcs, and arc coverage is not possible with rooted paths. Our algorithm still provides arc coverage of all those arcs that are reachable from the start node.

The above implementation suffers from some of the problems of recursion. Programs using the **TestGraphIterator** to traverse very large graphs may face memory problems due to memory intensive recursion. This problem might not occur if recursion were avoided. An early version of the **TestGraphIterator** class was developed using a non-recursive form of the above algorithm to traverse the graph. The access routine **next** traversed only one path at a time. This algorithm was unwieldy and complicated.

The recursive algorithm works correctly without problems for testgraphs used in the testgraph class test methodology, where very large graphs are not normally needed. Therefore, it was decided to use the recursive version of the algorithm in the final **TestGraphIterator** class.

Chapter 5

Testing The TestGraph Classes

The `TestGraph` classes have been tested using the testgraph methodology. A single test suite was developed to test the `TestGraph` and `TestGraphIterator` classes.

5.1 The test strategy

A testgraph was created to test the `TestGraph` classes, and is referred to as the "driver testgraph" through this chapter. Each node from this testgraph represents a testgraph with a specific graph "pattern" formed by its nodes and arcs. The test suite places each of these testgraph patterns into a `TestGraph` object, one at a time, and checks the `TestGraph` object state, the paths generated from it by a `TestGraphIterator` instance, and all exceptions signalled from either class.

5.1.1 Using `TestGraph` classes to drive `TestGraph` testing

In keeping with the testgraph methodology, this test suite uses objects of the `TestGraph` classes to drive the testing, henceforth called "driver `TestGraph` objects". This test suite is unusual in that the CUT itself is used as part of its test driver.

Therefore, the possibility exists that faults in the CUT could be hidden during testing, or spurious faults reported, due to the driver `TestGraph` objects. It may also be that arc coverage is not achieved for the driver testgraph, if the driver `TestGraph` objects malfunction, resulting in some portions of the test suite not being executed at all.

However, in practice, using the `TestGraph` classes to drive the testing has not turned out to be a hindrance to proper testing. The functionality provided by the driver `TestGraph` objects was manually checked. Node and arc labels from paths traversed through the driver testgraph, printed out as the test suite executes, were inspected for validity. Each path generated by the test driver was inspected to be a valid path from the driver testgraph. It was ascertained that each path began at the start node, that paths generated were traversable from it, and that arc coverage was achieved.

The driver `TestGraph` objects are not used for test output checking, or for any direct testing of the CUT. Instead, they are used only to load and traverse the driver testgraph. Since the role of the driver `TestGraph` objects is limited to storing a testgraph and generating paths through it, and these paths have been manually checked for correctness, it is assumed that the driver `TestGraph` objects do not interfere with the testing of the CUT.

5.1.2 The Test Plan

Figure 5.1 shows part of the Test Plan for the `TestGraph` classes. The test suite parameter, `length`, is used to calculate the number of nodes in the testgraph stored in the CUT. Test cases are selected based on the class interface specifications and the class implementations. Normal case behaviour is tested by storing testgraph patterns in a `TestGraph` object, and applying the interval rule on the size of the testgraphs. The testgraphs stored include a testgraph with no nodes and arcs, one having a

```
TEST SUITE PARAMETERS
    length: integer in [0,MAXNODES]
    selected values: <0,1,3,10,MAXNODES>

...

TEST CASE SELECTION STRATEGY
Specification-based tests
    Normal case
            Special values for object state
                    interval rule on the size of testgraphs stored
            Special values for member function parameters
                    interval rule on add order for get calls
    Exceptions
            Generate each exception at least once

...
```

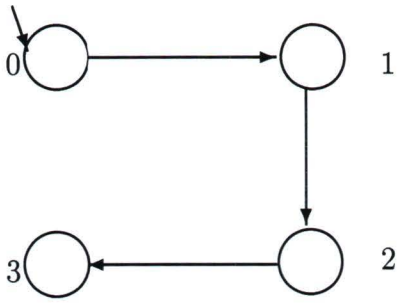
Figure 5.1: `TestGraph`—Test Plan

single node, some having all possible nodes and arcs, and others in between for the current test suite parameter. `TestGraphIterator` instances are created and paths are generated, and checked for correctness using the test oracle. Each exception is generated at least once from each place where it can be generated.

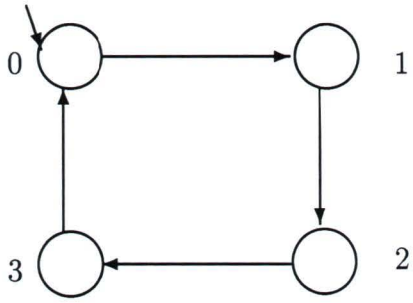
5.1.3 The pattern testgraphs

Testgraph patterns stored in the CUT should be such that they test the `TestGraph` classes thoroughly. The patterns selected represent testgraphs whose nodes and arcs can be generated by relatively simple programs. These patterns together require non-trivial `TestGraph` construction and traversal, without too heavy an implementation cost.

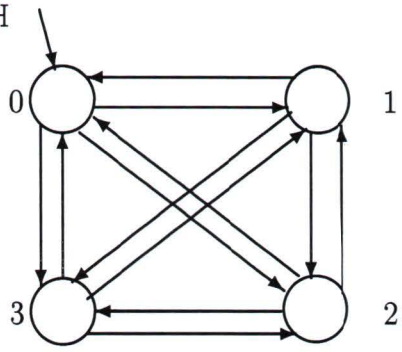
The pattern testgraphs used here are the `EMPTY`, `CHAIN`, `RING`, `COMPLETE`, and `STAR` testgraphs, shown in Figure 5.2. Arcs for all testgraph patterns are labeled with the sum of their source and destination node labels. The `EMPTY` testgraph is the empty



A CHAIN GRAPH



A RING GRAPH



A COMPLETE GRAPH

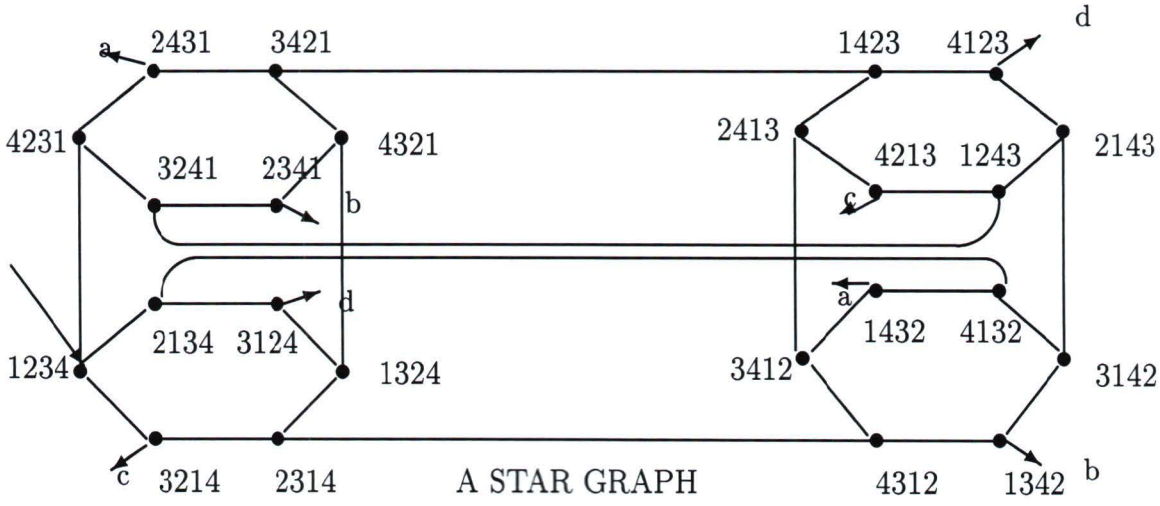


Figure 5 2 Testgraph patterns (test suite parameter = 4)

graph A CHAIN testgraph having n nodes is a graph such that for all $i \in [0, n-2]$, there exists an arc from node i to node $i+1$, and these are the only arcs in the graph A RING testgraph of n nodes is a CHAIN graph with an arc from node $n-1$ to node 0, as shown in Figure 5 2 In the COMPLETE testgraph there is an arc from each node to every other node

In a STAR graph of order n, the label for each node is a permutation of the digits 1,2, ,n The start node is labeled 12 n Each node is connected to $n-1$ other nodes For example, if the test suite parameter (length) is 4, the start node for the STAR testgraph is 1234, as in Figure 5 2 Other nodes with arcs to and from a node in the testgraph are obtained by interchanging the first digit of the label with its other digits In the above example, other nodes with arcs to and from the start node for the STAR testgraph are 2134, 3214, and 4231 STAR testgraphs allow us to test the TestGraph classes with different node labels than the ascending integers for other testgraphs

The set of nodes in an EMPTY testgraph is the empty set, and that for a RING, a CHAIN, or a COMPLETE testgraph is [0 length-1] The number of nodes in a STAR testgraph is the factorial of length When the factorial of length is greater than the largest number of nodes that can be stored in a TestGraph object (MAXNODES), the STAR testgraph behaves like an EMPTY testgraph Currently MAXNODES is 50, therefore STAR testgraphs of length more than 4 cannot be stored, and behave like EMPTY testgraphs

A STAR graph is defined for undirected graphs The TestGraph class, however, stores only directed graphs Storing the STAR graphs in the TestGraph class requires conversion of undirected STAR graphs into directed testgraphs A STAR testgraph, as implemented in this test suite, has the same nodes as an undirected STAR graph of the same arity If nodes a and b in an undirected STAR graph of arity n have an arc between them, then the STAR testgraph of arity n has a directed arc from node a to

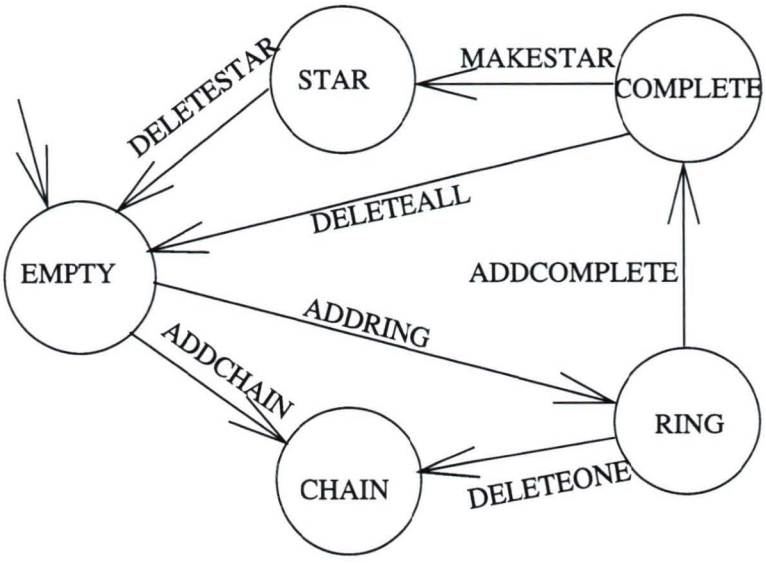


Figure 5.3: `TestGraph`—testgraph

node b and another from node b to node a.

5.1.4 The driver testgraph

The driver testgraph is shown in Figure 5.3. The `EMPTY` testgraph is the start node. The `CHAIN` and `RING` nodes test the CUT when a single path provides arc coverage of the testgraph stored in the CUT, and `COMPLETE` represents the maximal test case for each test suite parameter. The transition `ADDRING` is achieved by adding all possible nodes for the current test suite parameter, and arcs by definition of `RING` graphs, to an `EMPTY` testgraph. The `CHAIN` testgraph is generated from a `RING` by deleting one arc, by the transition from the `RING` testgraph to the `CHAIN` testgraph, called `DELONE`. The `COMPLETE` testgraph is generated by adding arcs so that there is an arc from every node in the testgraph to every other node in it, by the transition `ADDALL`. A `STAR` testgraph is generated from the `COMPLETE` testgraph by deleting all nodes and arcs, and adding those for `STAR` graphs by the transition `MAKESTAR`.

5.2 The test suite

With the test strategy in place, we proceed to the test suite itself In the testgraph methodology, a test suite consists of an oracle and a driver The test oracle for the `TestGraph` test suite is composed of the pattern implementation and the `Oracle` class The implementation for creating the testgraph patterns is a set of standalone routines in the file `pattern C` The test driver is composed of the `Driver` class and scaffolding code, as usual

5.2.1 Oracle strategy

Testgraph patterns are created by invoking general purpose access routines, declarations for which are shown in Figure 5 4 If a testgraph pattern, p, is to be created when the test suite parameter is l, the call `nodeSet(`p`,`l`)` is used by the test oracle to create the set of nodes, n, and `arcSet(`p`,`l`,`n`)` is used by the oracle to obtain the set of arcs in the testgraph The call `adjacent(`p`,`l`,`s`,`d`)` returns true if there exists an arc from node s to node d in the testgraph with pattern p for test suite parameter l Special functions used for creating star graphs include the function `flip`, which swaps the first digit of a multidigit number with any one of its other digits, in order to create node labels for star graphs

Using the pattern routines, the `Oracle` class generates the nodes and arcs of the testgraph in the CUT, and stores them in its instance variables The `Oracle` class provides set calls to change the testgraph pattern currently represented by its instance variables, and get calls to allow the `Driver` to check the CUT state `Oracle` member functions test if a path obtained using a `TestGraphIterator` object is a path from the testgraph currently stored in the CUT, and if arc coverage has been achieved by a set of path

```
// ***public pattern functions***

ListSet<int>&     nodeSet(int,int),
ListSet<ArcInfo>& arcSet(int,int,ListSet<int>&),
int               adjacent(int,int,int,int),
int               flip(int,int,int),

// ***private pattern functions***

static void createStar(int,int),
static int  check(int),
static int  starToRing(int),
int         neighbour(int,int,int),
```

Figure 5.4 The pattern routines

5.2.2 Oracle implementation

The pattern routine implementations

Sets of nodes and arcs for the patterns are calculated according to pattern definitions. In the `nodeSet` function, `RING`, `CHAIN`, and `COMPLETE` testgraph pattern nodes are easily calculated using a loop. Node labels for `STAR` graphs are calculated in `nodeSet` by invoking the local function `createStar`, which internally uses `flip` and local functions `check` and `neighbour` to create the nodes. The `arcSet` routine iterates over all the nodes in the node set and uses `adjacent` to check if an arc should be present between any two. If so, the arc label is set to be the sum of the node labels and the arc is added to the set of arcs to be returned to the user. `adjacent` uses simple arithmetic to calculate if two nodes are adjacent, except in the case of a `STAR` graph, where local functions are used.

```
class Oracle {
public:
        Oracle(int);
        ~Oracle();

        void setGraph(int);
        int getNumNode() const;
        int getLabel(int,int) const; // expects node labels
        int getNode(int,int) const; // expects node/arc indices
        int isPath(Path*);
        int allDone() const;

        int node; // node identifier
        int length; // graph parameter
private:
        NodeType arcs[MAXNODES][MAXNODES]; // stores nodes and arcs
...
};
```

Figure 5.5 `Oracle`—syntax

The Oracle class implementation

An extract from the `Oracle` class interface syntax is shown in Figure 5.5. This class uses a two dimensional array, `arcs`, of a local type, `NodeType`, which stores a node or arc `label` and a `visited` flag indicating if the node or arc has been visited. `arc` stores nodes on the main diagonal, and arcs elsewhere. The `Oracle` class constructor parameter is the test suite parameter, `length`, used to calculate the nodes and arcs in the testgraph in the CUT. The constructor sets the current pattern in the `Oracle` object to be `EMPTY`, and resets all entries in `arcs`. The function `setGraph` is used to create a testgraph pattern and store its arcs and nodes in `arcs`. `setGraph` is a generalized routine, using pattern functions `nodeSet` and `arcSet` to create sets of nodes and arcs for the testgraph pattern, and placing these in `arcs`, freeing the memory for the node and arc sets returned by the pattern functions.

The function `getNumNode` returns the number of nodes expected in the CUT. The call `getExistNode(`i`)` returns true if i is expected to be a node in the CUT. The call `getLabel(`i`,`j`)` returns the label for the arc that is expected to exist from node i to node j in the CUT, and `getNode(`i`,`j`)` returns the label for the node or arc expected to be in the CUT from the node at position i to the one at position j in `arcs`.

The call `isPath(`p`)` checks if the path, p, is a valid path from the testgraph stored in the CUT. `isPath` checks that the first element in the path is the start node, and then iterates over all elements in the path, checking that each node and arc in the path is expected to be in the CUT, and setting its `visited` flag in `arcs`. `isPath` also checks that, for each arc in the path, the previous node in the path is its source, and the next node in the path its destination, as expected.

The function `isPath` does not check that all arcs and nodes in the `TestGraph` CUT have actually been visited in the paths provided by the iterator. This is done using `allDone`, which returns true if all arcs and nodes in the testgraph stored in the CUT have been visited. This function iterates over all valid elements in `arcs`, checking that every node and arc in the array has its `flag` set. If any of the nodes or arcs are unreachable from the start node, `allDone` returns false.

5.2.3 Driver strategy

The `Driver` class provides functions to be executed whenever a node or arc in the driver testgraph is traversed. A `Driver` object uses the `Oracle` to obtain the nodes and arcs that should be in the CUT at each driver testgraph node, and provides routines to change the CUT state accordingly. The `Driver` class stores a `TestGraph` object and an `Oracle` object as instance variables. `Driver` member functions use the information provided by the `Oracle` class to check for node and arc existence in the `TestGraph` object, so as to check the CUT state. Paths in the testgraph stored in the

CUT, traversed using `TestGraphIterator` instances, are checked to be valid paths by invoking `Oracle` functions. Exceptions for the `TestGraph` and `TestGraphIterator` classes are tested to be correctly raised. Most exception-checking code is invoked after the normal case checking code.

5.2.4 Driver class implementation

Figure 5.6 shows an extract from the `Driver` class syntax. The public functions `node`, `arc`, and `reset`, and the private instance variables `cut` and `orc` perform their usual functions. The function `reset` is used whenever a new path in the driver `TestGraph` is to be traversed. Functions `arc` and `node` are executed each time an arc or node in the driver `TestGraph` is traversed. The `Driver` class takes the test suite parameter as a constructor parameter, and passes it to the `Oracle` constructor.

The `arc` function is implemented as a `switch` statement on the arc being traversed in the driver testgraph. This function modifies `orc`, using `Oracle.setGraph`, to have the pattern given by the destination node of the driver testgraph arc being traversed. The state of the CUT is tested when `node` is executed, and any CUT errors that may have occured in the `arc` call show up at that time.

The `cut` state and `TestGraphIterator` class behaviour are checked in `checkCut`. This function first tests `TestGraph.getNumNode` against the number of nodes in the CUT, as reported by `orc.checkGraph` is then used to test other `TestGraph` get calls. This function iterates over those nodes and arcs that `orc` expects to be in `cut`, and checks if they are indeed present, by invoking `TestGraph` get calls. `checkCut` then invokes `checkStartNode` which sets and checks the start node for the CUT using the relevant `TestGraph` functions.

Before setting the start node in order to traverse the testgraph in the CUT, the `checkNoStartExc` function is invoked to check that exception `TGNoStartExc` is cor-

```
class Driver : public AbstractDriver {
public:
        Driver(int);
        void reset();
        void arc(int);
        void node();
private:
        TestGraph cut;
        Oracle orc;

        // cut modifiers
        void changeCut(const Oracle&,const Oracle&);
        void makeNodes(const Oracle&,const Oracle&,int);
        void makeArcs(const Oracle&,const Oracle&,int);
        void callAddArc(int,int);
        void callDelArc(int,int);

        // cut state checking functions
        void checkCut(TestGraph&,Oracle&);
void checkGraph(TestGraph&,Oracle&) const;
        void checkStartNode(TestGraph&,Oracle&);

        // exception checking functions
        void checkExistNodeExc(TestGraph&,Oracle&);
        void checkExistArcExc(TestGraph&,Oracle&);
        void checkFullExc(TestGraph&,Oracle&);
        void checkNotExistNodeExc(TestGraph&,Oracle&);
        void checkNotExistArcExc(TestGraph&,Oracle&) const;
        void checkNoStartExc(TestGraph&,Oracle&);
        void checkEndExc(TestGraph&,TestGraphIterator&,Oracle&) const;
};
```

Figure 5.6: Driver—syntax

rectly signaled. The exception is generated by adding a node to the CUT, making it the start node, deleting it, and then invoking the `TestGraphIterator` constructor and `reset`. The `startNode` then is a node which no longer exists in the CUT, and the exception should get signalled from both these functions.

Paths generated by `TestGraphIterator` `next` are tested using the `Oracle` class routine, `isPath`. A new `TestGraphIterator` object is created and the start node is set. `next` is repeatedly invoked on this object, in an iterative loop, and each path thus generated is checked to be correct. Iteration terminates once the `TestGraph-Iterator` function `isEnd` returns true. The `Oracle` class function `allDone` is used to make sure that some arc or node has not been traversed while `isEnd` returns false, and that they have all been visited once `isEnd` returns true. At this stage, exception `TGEndExc` is tested by invoking `checkEndExc`. This function invokes `next`, and since all paths have already been traversed, the iterator should signal `endExc`.

Once the normal case behaviour has been checked, the rest of the exceptions are tested. `checkExistNodeExc` tests exception `TGExistNode` by applying the interval rule in attempting to add an existing node to the CUT. For all but `STAR` graphs, attempts are made to add the start node, the final node, and one node in between. For `STAR` graphs, attempts are made to add the start node and one other node.

The `checkExistArcExc` function tests exception `TGExistArcExc`, generated by applying the interval rule while adding existing arcs to the CUT. `checkFullExc` tests exception `TGFullExc`, which can only be raised when `MAXNODES` nodes have already been added to the CUT, and another node is attempted to be added.

The exception `TGNotExistsNodeExc` is tested to be signaled using `checkNot-ExistNodeExc`. An attempt to delete a non existent node from the CUT, or an attempt to add an arc from, or to, a non existent node in the CUT should raise the exception. Finally, the `TGNotExistArc` exception is tested to be signaled from `checkNotExistArcExc`, by invoking `TestGraph` `getLabel` for invalid nodes.

Chapter 6

Testing The LEDA graph class

The testgraph methodology has been used to test several classes developed by us. In order to demonstrate the applicability of this methodology for industrial classes, we needed to test some commercial classes. We have chosen to test a widely used graph class from a library of classes called LEDA. We first introduce LEDA and its graph classes, and then explain their testing.

6.1 LEDA

LEDA (Library of Efficient Data Types and Algorithms) is a software library consisting of C++ classes for many different abstract data types, and algorithms for them. The library provides short, semi-formal specifications for each function. These specifications are abstract and are written as English prose. Many LEDA data types are provided as parameterized classes.

LEDA was developed at the Max Planck Institute for Computer Science, Germany, by Naher and Mehlhorn, [17], [16] as part of an ongoing project, begun in 1989. The library is available at an FTP (File Transfer Protocol) site. We have used LEDA

version 3 1 2 , and the source code and manual have been obtained from the FTP site

6.1.1 The graph class

LEDA provides the **graph** class, which implements directed and undirected multi-graphs An instance of **graph** contains a list of nodes and a list of arcs Each node has associated with it, an adjacency list of the out-arcs from it, and an adjacency list of the in-arcs to it Each arc is in the list of out-arcs from its source node and in the list of in-arcs to its destination node It is not possible to specify arc or node labels The parameterized **GRAPH** class allows information to be associated with nodes and arcs, and supports most public functions of the **graph** class, apart from some more

Services offered by the graph class

The **graph** class provides approximately a hundred routines, specifications for which are in the LEDA manual [19] Apart from set calls to build graphs, and get calls to retrieve info about the graph, algorithms for graph and network traversal are provided, and two types of iterators are available

Set calls create or delete nodes and arcs For example, `hide_edge(`e`)` is used to temporarily disconnect an arc from its source and destination nodes, leaving the arc in the graph, and `restore_edge(`e`)` can be used to re-connect it It is possible to interchange source and destination nodes for arcs, or for every arc in the graph, to insert an arc from the destination of the arc to its source Methods are available to sort the elements in the node and arc lists, in place

The get calls in the **graph** class provide out and in degrees for a node the number of in-arcs or out-arcs for a node There are methods which return the source or destination node for an arc, e g , the function `opposite` provides the other node

connected to an edge if one of them is known. Member functions of the class also provide the number of nodes and arcs, or the list of all nodes or arcs in the graph. Other methods provide the list of out arcs or in arcs to a node in the graph, or traverse the adjacency lists for each node. Functions are provided to obtain the first and the last node or arc in the lists in the graph, or to obtain a successor or predecessor of a node or arc. For example, `cyclic_adj_succ(`e`)`, provides the cyclic successor of arc e in the adjacency list of the source node of e.

Iterators for adjacency lists in the **graph** object are called "adjacency iterators", and they provide operations to reset the iterator and to obtain the next element from it. For example, the call `next_adj_edge(`e`,`n`)`, places the next edge from the adjacency list of node n, into the edge e. Iteration is also provided as a set of macros which iterate over nodes or arcs in a graph. Iteration macros provide a loop like the C++ `for` statement, except that the graph cannot be modified inside the loop. An iteration macro extracts elements from the graph one at a time, repetitively executing the statements in the loop. For example, `forall_nodes(`n`,`g`)` { s } executes the C++ statement s, with each node in graph g assigned once to node n.

The **graph** class provides functions to read and write graphs to streams. Functions can read or write complete graphs as well as single nodes or edges. For example, the call g `write(`o`)` writes out the nodes and arcs for a graph g to output stream o.

6.1.2 The GRAPH class

`GRAPH` is a template based, directed graph class, parameterized with node and arc types. For example, `GRAPH<`n`,`a`>` is a `GRAPH` object with nodes of type n and arcs of type a. Information is associated with nodes and arcs using set calls, and is retrieved using get calls. For example, if g is a `GRAPH` object, with integer node and arc labels, then g `new_node(`n`)` adds a new node with integer label n to g. It

is also possible to sort nodes or edges according to the ordering defined on their labels, and to read or write node and arc labels in a graph. Algorithms provided for traversal of graphs in this class are depth-first search and breadth-first search. Other algorithms are provided for computing the connected components, etc., apart from network algorithms.

6.2 GRAPH class test strategy

Having introduced the LEDA graph classes, we now describe the strategy for testing these classes. We created a test suite for the `GRAPH` class, which inherits publicly from the `graph` class, and stores directed graphs. The `GRAPH` test suite also tests member functions of the `graph` class. This test suite made extensive use of the `TestGraph` test suite, and was developed with far less effort than might otherwise have been needed.

The test strategy for the `GRAPH` class is similar to that used to test the `TestGraph` classes. A testgraph (the driver testgraph) was created, with each node in it representing a graph pattern in a `GRAPH` instance. The patterns stored were chosen to exercise the `GRAPH` class thoroughly. The test suite places each of these patterns into a `GRAPH` object, checking its state and invoking CUT functions to check graph traversal.

Most of the `GRAPH` member functions were tested in this test suite. An attempt was made to test a representative sample of functions: a few from each family of access routines, such as set/get calls, iterators and traversal algorithms. Since the `GRAPH` class stores directed graphs, functions for undirected graphs were not tested. We decided not to test those functions in the `graph` class which are overridden by `GRAPH` class functions, since testing the latter was sufficient to demonstrate the graph class test scheme. Most functions in the classes are very similar, and it would not be a major problem to test each and every function, with the infrastructure in place.

Out of eighteen functions defined directly for the `GRAPH` class, fourteen were tested.

graph class functions tested included set calls to build and destroy graphs. All get calls, apart from iterators, were tested. Each adjacency iterator routine was tested, but some graph iterator macros were not. Two algorithms for graph traversal were tested. In toto, seventy two, out of one hundred, functions were tested.

6.2.1 The graph patterns

Patterns used in the GRAPH testing were the same as the ones used for TestGraph testing: EMPTY, CHAIN, RING, COMPLETE, and STAR graphs. These patterns were generated using the pattern routines described in Chapter 5. There were changes to the way the pattern routines are used by the test oracle, but none to the pattern routines themselves.

The GRAPH class test suite had to test multigraphs, and so the patterns represent directed multigraphs. If the pattern routines specify that a graph pattern has an arc from source node s to destination node d, the corresponding multigraph was generated by the test oracle by adding multiple arcs from s to d, each having a different arc label. This is done in the test oracle and the pattern routines were not modified.

6.2.2 The Test Plan

The Test Plan for the GRAPH class is similar to the TestGraph Test Plan. The test suite parameter, length, is a measure of the size of the graph. The selection of special values for GRAPH member function parameters is based on to the specifications of the functions. An interval rule on add order is applied for all relevant calls. Program-based testing was not done in this case, since specifications-based testing seems sufficient to demonstrate our test scheme, and program-based testing would involve testing the highly complex implementation of the GRAPH class, with not much gain. The GRAPH class does not support exceptions, and so no exception testing was

done The driver testgraph used here is the same as that used in the `TestGraph` testing, but each node of this driver testgraph represents a directed multigraph

6.2.3 `TestGraph` testing vs. GRAPH testing

Testing the `GRAPH` class has had its similarities to as well as differences from the `TestGraph` testing The framework for the two test suites is very similar, but there have been some changes to both the `Driver` and the `Oracle` classes `GRAPH` testing tests multigraphs, and tests a far larger number of CUT member functions The traversal algorithms tested here are different from the rooted path traversal provided by the `TestGraphIterator` class There are no exception tests performed here, and we cannot use code coverage as a valid criterion to measure the testing, since testing is only based on the specifications

6.3 The test implementation

With the test strategy in place, we now go on to the test suite implementation The test suite consists of the `Oracle` and the `Driver` class implementations, and the pattern routines The pattern routines have not changed, and hence are not described here

6.3.1 The Oracle class

Oracle strategy

The `Oracle` class used here has few changes from the `TestGraph Oracle` other than supporting multigraphs Labels for each arc in the multigraph are not actually stored in the `Oracle`, and most are calculated as needed Whenever a graph pattern has an arc l from source node s to destination node d, the multigraph in the `Oracle` class

```
// ***contants***
const int DEPTH = 2,

// ***types***
struct NodeType {
        int value,
        int visited[DEPTH],
},

// ***class declaration***
class Oracle {
public

        int getNumArc() const,
        int getLabel(int,int,int) const, // expects node labels
        int getNode(int,int,int) const, // expects node/arc indices
        int getPos(int) const,
        int allNodesDone() const,
        int isEdge(int,int,int) const,
        int isNode(int) const,

},
```

Figure 6.1 `Oracle`—syntax

has two arcs from s to d, one with label l and the other with label $2 \times l$. The label for arc label l is stored, and labels for other arcs are calculated as needed. Information is stored about whether or not each arc has been visited.

Oracle implementation

Figure 6.1 shows that part of the `GRAPH Oracle` interface syntax which is different from the `TestGraph Oracle` interface syntax. If a graph pattern has an arc from source node n to destination node d, the constant `DEPTH` is the number of parallel arcs from s to d. The `NodeType` structure now stores the label for a single arc, and `visited` flags for `DEPTH` number of arcs, indicating whether or not each arc has been

visited `arcs` is an array of `NodeType`, storing nodes on the main diagonal and arcs elsewhere.

Most member functions in this class are the same as for the `TestGraph Oracle`. The new function `getNumArc` returns the number of arcs expected in the CUT. `getNode(`s`,`d`,`i`)` returns the label of the i'th arc from node s to node d, and the call `getLabel(`s`,`d`,`i`)` returns the label of the i'th arc from the node stored in `arcs` at position s to the one stored at position d. `isEdge(`s`,`d`,`l`)` returns true if there exists an arc l, from node s to node d in the CUT. `isNode(`n`)` returns true if node n exists in the CUT, and `allNodesDone` returns true if every node in the graph has been visited.

6.3.2 The Driver class

Driver strategy

The `Driver` class obtains graph patterns from the `Oracle` class, and uses these to load graphs into a `GRAPH` instance. Each `Driver` function tests a family of similar CUT functions. There is no exception-checking code, and only legal calls are made on the CUT. Graph traversal is tested by invoking the relevant CUT routines and then using the `Oracle` class to ensure that all nodes and arcs have actually been traversed.

Driver class implementation

Figure 6.2 shows part of the `Driver` interface syntax. Public functions `node`, `arc`, and `reset`, (not shown) perform the same functions as in the `TestGraph Driver`. The variable `cut` is a `GRAPH` instance having integer nodes and arcs, and `orc` is an `Oracle` instance. The local utility function `findVertex` extracts a specific node from a `GRAPH` instance, and `findEdge` extracts a specific arc from it. The utility function call `findVal(`i`,`a`,`s`)` returns the position of integer i in an integer array a of size s.

```
// *****interface syntax*****
class Driver  public AbstractDriver {

private
        GRAPH<int,int> cut,
        Oracle orc,

        // cut modifier functions
        void changeCut(const Oracle&,const Oracle&,GRAPH<int,int>&),
        void makeNodes(const Oracle&,const Oracle&,int,GRAPH<int,int>&),
        void makeArcs(const Oracle&,const Oracle&,int,GRAPH<int,int>&),
        void callAddArc(int,int,int,GRAPH<int,int>&),
        void callDelArc(int,int,int,GRAPH<int,int>&),

        // cut check routines
        void checkDegree(GRAPH<int,int>&,Oracle&),
        void checkIterators(GRAPH<int,int>&,Oracle&),
        void checkAll(GRAPH<int,int>&,Oracle&),
        void checkLists(GRAPH<int,int>&,Oracle&),
        void checkInOut(GRAPH<int,int>&,Oracle&),
        void checkUpdate(GRAPH<int,int>&,Oracle&),
        void checkPaths(GRAPH<int,int>&,Oracle&),

        // local functions
        void checkAdjacent(int *,int *,int,int,node),
        void checkIn(int *,int,node),
        int checkTraversal(list<node>&),

        // local variables
        int sumNodes,
        int sumEdges,
},

// ***local utility function declarations***
static node findVertex(GRAPH<int,int>&,int),
static edge findEdge(GRAPH<int,int>&,int,int,int),
static int findVal(int,int*,int),
```

Figure 6.2: Driver—syntax

The CUT modifier routine `changeCUT(s,d)`, invoked with a source `Oracle` instance s and a destination `Oracle` instance d, modifies the CUT from the state represented by s to that represented by d. This function uses other functions such as `makeNodes`, which are close to similarly named functions in the `TestGraph Driver`.

The function `checkDegree` is used to check the CUT access routines associated with the in-degree and out-degree of nodes, and to collect data used to test other functions. `checkDegree` iterates over all nodes in the graph, and invokes `Driver` routines `checkAdjacent` and `checkIn`, for each node. `checkAdjacent` tests the family of CUT routines for arcs and nodes adjacent to a node. Arcs are adjacent to their source nodes, and nodes are adjacent to the destination nodes of their out-arcs. CUT routines which cyclically provide the adjacent successor or predecessor of an arc in the adjacency list of a node, are tested by going through the complete adjacency list several times. The `Driver` function `checkIn` tests the family of CUT routines concerned with in-arcs to a node, such as `in_edges` and `first_in_edge`. These functions are tested the same way as functions for adjacent arcs.

The routine `checkIterators` initializes the adjacency iterators of the CUT and checks the nodes and arcs obtained from them. One macro iterator is tested in the function `checkDegree`, the others are tested here by checking that the nodes and arcs obtained are in the CUT.

The function `checkAll` retrieves node and arc lists from the CUT and checks their sizes and elements in the lists. The function `checkLists` obtains each element from the node and arc lists in the CUT, and verifies that each such node and arc is in the CUT. Elements are obtained by traversing the lists in both forward and reverse order and it is checked that the same elements are returned both ways. `checkInOut` reads in and writes out graphs, and the output is verified manually.

The `checkUpdate` function checks the set calls in the CUT. All arcs in the CUT are deleted and re-inserted, and then hidden and restored. A copy of the CUT is made,

and the CUT function `rev` is invoked to reverse all its edges, which is then verified CUT routines to delete all edges and nodes are tested by invoking the routines and checking that the CUT is empty

Graph traversal is tested by the `Driver` function `checkPaths`, which invokes the CUT function `DFS` to perform a depth first search, and `BFS` to perform a breadth first search The local function `checkTraversal` is invoked, to test the list of nodes obtained from the traversal Each node is checked to be in the CUT, and the `Oracle` function `allNodesDone` is used to ensure that each node in the CUT has been visited

6.4 Failures in the CUT

Several failures were detected in the `GRAPH` class, and reported to the LEDA developers The LEDA developers have acknowledged these, and expect to provide bug fixes in future LEDA releases

A failure was detected in `cyclic_in_succ(`e`)`, which should cyclically return the successor of arc e from the list of in-arcs for e's destination, n This function, after first returning all elements in the list of in arcs of n, starts returning edges from the list of out arcs of n This failure was also detected in `cyclic_in_pred(`e`)`, which should return the cyclic predecessor of the e in the list of in arcs of e's destination node, n This function too starts returning arcs from the out arcs of n, after first returning each in arc of n

Another failure detected was in the function `in_edges(`n`)`, which should return the list of in-arcs for node n This function generated linker errors, since the function was not implemented at all

The function `adj_nodes`, expected to return the nodes adjacent to a node, was also detected to have a failure If a graph has n ($n > 1$) arcs from a node s to another node d, `adj_nodes(`d`)` returns n copies of node s, instead of one

Chapter 7

Conclusion and Future Work

7.1 Summary and Conclusion

Much work remains to be done in the area of object-oriented testing in general, and that of class testing in particular. Tools and techniques are needed to provide a framework for testing software developed in object-oriented languages, such as C++. Such techniques need to be demonstrated to be feasible, both in a controlled, laboratory environment, and in a real-life, industrial setting.

We have developed a graph based tool for the automated testing of C++ component classes by the testgraph methodology. Part of this tool has been implemented in C++ by the `TestGraph` and `TestGraphIterator` classes, which store simple, directed testgraphs. A traversal mechanism is provided, which performs rooted traversal and provides arc coverage of the testgraph, if at all possible.

The `TestGraph` classes are used in the testing of C++ classes, with nodes in the testgraph representing states in the CUT, and arcs representing transitions between these states. The testgraph methodology provides a complete test method which involves loading and traversing a testgraph stored in the `TestGraph` class, modify-

ing the CUT state when traversing each testgraph arc, and testing the state when traversing each testgraph node.

We present an easily reproducible scheme for testing C++ graph classes, which follows the testgraph methodology. It stores special graphs into the graph CUT, and checks that they are correctly stored. Each graph is traversed, and the traversal verified. This scheme was implemented to test the `TestGraph` classes, and certain test criteria, such as 100% statement coverage, were achieved.

The graph class testing scheme was then demonstrated in an industrial setting by testing the LEDA `GRAPH` class, which implements directed multigraphs. While this class proved to be much larger and more complex than the `TestGraph` classes, the graph testing scheme was easily adapted to test most methods in it. It also provides different graph traversal mechanisms, which were tested to work correctly.

The output log-file generated from the `TestGraph` test suite was approximately 570Kb in size, and had about 18.5 thousand lines, while that generated from the LEDA test suite was about 43Mb in size, and had over 3.3 million lines of output. Roughly speaking, each line of output in the log-file corresponds to one invocation of a CUT method, and therefore approximately 18.5 thousand `TestGraph` method invocations were made, and approximately 3.3 million LEDA `GRAPH` method invocations were made.

Our graph test scheme reported failures in the LEDA `GRAPH` class. The first version of this class was released in 1991, and it has been widely used around the world since then. Therefore, the reporting of failures in the class, when tested using our graph testing scheme, shows the need for systematic techniques for testing such component classes. It also demonstrates that the graph testing scheme developed by us works correctly for medium-sized classes in an industrial setting.

In summary, the contributions of my thesis research work have been: the development of the `TestGraph` and `TestGraphIterator` classes to form the basis of the test-

graph methodology, the development of a graph class testing scheme using the test-graph methodology, and the demonstration of this scheme to testing the `TestGraph`, the `TestGraphIterator`, and LEDA `GRAPH` classes.

7.2 Future Work

The `TestGraph` classes store and traverse testgraphs. Since multigraphs are currently not supported, one transition from a CUT state, s, to a CUT state, d, cannot be distinguished from another transition from s to d, as both are represented by the same arc. For example, in Figure 3.4, which shows the testgraph for the `IntSet` class, the arc `ADDALL` represents the transition adding integers to the empty set to make a full set. If the `IntSet` implementation uses a sorted array to store the set, the tester may wish to add the same elements to an `IntSet` object, more than once, in different orders, for instance, once in an ascending order and once in a random order. Therefore, there will be two different transitions from node `EMPTY` to node `ALL`, which need to be represented by two different arcs. The arc `ADDALL`, from `EMPTY` to `ALL`, can represent the transition when elements are added sequentially to the set. The arc `ADDALLRANDOM`, again from `EMPTY` to `ALL`, can represent the transition when the same elements are added to the set in a random order.

Therefore, the `TestGraph` classes can be modified to support multigraphs, where each arc from one node to another will represent a different way of making that transition. The traversal algorithm must also be modified, in order to provide arc coverage for multigraphs. The `TestGraph` test suite will then need to be modified to test multigraphs. However, the LEDA `GRAPH` test suite already tests for multigraphs, and the testing scheme used there can be used in the `TestGraph` test suite. The graph traversal testing scheme may also need to be modified to test the new traversal algorithm.

The current traversal algorithm for the `TestGraph` classes provides arc coverage by producing simple rooted paths through the testgraph. This algorithm can be replaced by other algorithms, such as the Maximal Simple Rooted Paths (MSRP) algorithm, which produces every maximal non-cyclic path possible from the start node in the testgraph. Whereas the current algorithm terminates on achieving arc coverage, MSRP provides all possible simple rooted paths in the testgraph, hence producing many more test cases. This algorithm was implemented in a prototype of the `TestGraphIterator` class, but was not adopted in its final version, since memory problems due to recursion (of the sort described in Chapter 5) were faced when a huge number of paths were generated by the algorithm. The current algorithm produces a subset of the paths produced using MSRP, and appears to be sufficient for now, since it achieves arc coverage.

Bibliography

[1] Thomas R. Arnold and William A. Fuson. Testing "In A Perfect World". *Communications of the ACM*, September 1994.

[2] Imran Bashir and A. Goel. Testing C++ Classes. In *Proceedings of International Conference on Software Testing, Reliability, and Quality Assurance*, 1994.

[3] B. Bezier. *Software Testing Techniques*. Van Nostrand Reinhold Company, Inc., 1990.

[4] Robert V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, September 1994.

[5] S. P. Fiedler. Object-Oriented Unit Testing. *Hewlett-Packard Journal*, 1989.

[6] Paul Townsend Gail C. Murphy and Pok Sze Wong. Experiences with Cluster and Class Testing. *Communications of the ACM*, September 1994.

[7] D. M. Hoffman. A Case Study in Module Testing. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society, 1989.

[8] D. M. Hoffman and XianHong Fang. Testing the CSet++ Collection Class Library. *CASCON'94*, 1994.

[9] D M Hoffman and Paul Strooper Graph Based Class Testing In *Proceedings 7th Australian Software Engineering Conference*, pages 85–91, 1993

[10] D M Hoffman and Paul Strooper Software Design and Maintenance A document driven approach To be published by International Thomson Computer Press, 1995

[11] Daniel Hoffman and Paul Strooper The Testgraph Methodology Automated testing of collection classes To appear in Journal of Object Oriented Programming, 1995

[12] IEEE IEEE standard for software unit testing *The Institute of Electrical and Electronics Engineers*, 1987

[13] Paul C Jorgenson and Carl Erickson Object-Oriented Integration Testing *Communications of the ACM*, September 1994

[14] Mary Jean Harrold, John D McGregor and Kevin J Fitzpatrick Incremental Testing of Object-Oriented Class Structures In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1990

[15] John D McGregor and Timothy D Korson Integrating Object-Oriented Testing and Development Processes *Communications of the ACM*, September 1994

[16] Kurt Mehlhorn and Stephan Naher Algorithm Design and Software Libraries Recent Developments in the Leda Project *Algortihms, Software, Architectures, Information Processing 92*, 1992

[17] Kurt Mehlhorn and Stephan Naher Leda A Platform for Combinatorial and Geometric Computing *Communications of the ACM*, January 1995

[18] Gail Murphy. Object-Oriented Systems Testing Methodology: Literature review. Technical Report No. TR92-0621, MPR Teltech, 1992.

[19] Stephan Naher. Leda manual. Technical Report MPI-93-109, Max-Planck-Institute for Informatics, 1993.

[20] Dewayne E. Perry and Gail Keiser. Adequate Testing and Object-Oriented Programming. *Journal of Object Oriented Programming*, 1990.

[21] Robert M. Poston. Automated Testing from Object Models. *Communications of the ACM*, September 1994.

[22] M. D. Smith and D. J. Robson. Object-Oriented Programming - the problems of validation. In *Proceedings of the 1990 IEEE Conference on Software Maintenance*, pages 272–281. IEEE Computer Society Press, 1990.

[23] M. D. Smith and D. J. Robson. A Framework for Testing Object-Oriented Programs. *Journal of Object Oriented Programming*, 1992.

[24] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley Publishing Company, 1991.

[25] C. D. Turner and D. J. Robson. Guidance for the Testing of Object-Oriented Programs. Technical Report No. TR 2/93, University of Durham, Durham, England, 1993.

[26] C. D. Turner and D. J. Robson. A suite of tools for the state-based testing of Object-Oriented programs. Technical Report No. TR 14/92, University of Durham, Durham, England, 1993.

Appendix A

The TestGraph classes

A.1 TestGraph Class Interface Specifications

```
// **********TestGraph interface specification**********

// ***custom abbreviations***
//        del for delete
//        ex for exists

// *****interface syntax*****

// ***constants***
const MAXNODES = 50,

// ***types***

// ***private information---begin***
typedef struct ArcType {
        int label,
        int exist,
},

// ***private information---End***

// ***class declarations***
class TestGraph {
public:
        friend class TestGraphIterator,
```

```
        TestGraph(int start = 0),

        void setStartNode(int), // node calls
        int getStartNode() const,
        void addNode(int),
        void delNode(int),
        int exNode(int) const,
        int getNumNode() const,

        void addArc(int,int,int), // arc calls
        void delArc(int,int),
        int exArc(int,int) const,
        int getLabel(int,int) const,

// ***private information---begin***
private:
        int node[MAXNODES], // nodes
        ArcType arc[MAXNODES][MAXNODES], // arcs
        int numNode, // number of nodes and arcs
        int startNode, // index of start node

        int findPos(int) const, // local function
},

// ***concrete state invariant***
//        (numNode in [0..MAXNODES]) and
//        there are no duplicates in node[0..numNode-1]

// ***abstraction function***
//        (|nodeSet| = numNode) and
//        nodeSet = {n | n in node[0..numNode-1]} and
//         arcSet = {<src,dst,label> | (exists i,j in [0..numNode-1])
//                                        (src = node[i] and dst = node[j]
//                                        and label = arc[i][j].label and
//                                        arc[i][j].exist)}

// ***private information---End***

// ***exception handlers***
void TGExArcExc(),
void TGExNodeExc(),
void TGFullExc(),
```

```
void TGNotExArcExc(),
void TGNotExNodeExc(),

/*****interface semantics---begin*****

***abstract state variables***
        nodeSet: set of integer
        arcSet: set of tuple of <src int,dst int,label int>
        startNode: integer

***abstract state invariant***
        |nodeSet| <= MAXNODES and
        (forall arc in arcSet)
                (arc.src in nodeSet and arc.dst in nodeSet) and
        not (exist arc1,arc2 in arcSet)
                (arc1.src = arc2.src and arc1.dst = arc2.dst)

***member functions***

TestGraph(node):
        transition: nodeSet,arcSet,startNode := {},{},node
        exceptions: none
        assumptions: none

setStartNode(node):
        transition: startNode := node
        exceptions: none
        assumptions: none

getStartNode():
        output: out := startNode
        exceptions: exc := none
        assumptions: none

addNode(node):
        transition: nodeSet := nodeSet+{node}
        exceptions: exc := (|nodeSet| = MAXNODES => TGFullExc
                          | node in nodeSet => TGExNodeExc)
        assumptions: none

delNode(node):
        transition: nodeSet,arcSet := nodeSet-{node},
```

```
                    arcSet-{<src,dst,label> in arcSet | src = node
                    or dst = node}
        exceptions  exc  = (not node in nodeSet => TGNotExNodeExc)
        assumptions  none

exNode(node)
        output  out  = (node in nodeSet)
        exceptions  none
        assumptions  none

getNumNode()
        output  out  = |nodeSet|
        exceptions  none
        assumptions  none

addArc(src,dst,label)
        transition  arcSet  = arcSet+{<src,dst,label>}
        exceptions  exc  = (exists label1)(<src,dst,label1> in arcSet =>
                            TGExArcExc |
                            not {src,dst} in nodeSet => TGNotExNodeExc)
        assumptions  none

delArc(src,dst)
        transition  arcSet  = arcSet -
                    {arc in arcSet | arc src = src and arc dst = dst}
        exceptions  exc  = ((not exists label)(<src,dst,label> in arcSet)
                             => TGNotExArcExc)
        assumptions  none

exArc(src,dst)
        output  out  = (exists label)(<src,dst,label> in arcSet)
        exceptions  none
        assumptions  none

getLabel(src,dst)
        output  out  = label such that <src,dst,label> in arcSet
        exceptions  exc  = ((not exists label)(<src,dst,label> in arcSet)
                            => TGNotExArcExc)
        assumptions  none

*****interface semantics---end*****/
```

A.2 TestGraphIterator Class Interface Specifications

```
#include "ListSetIter.h"
#include "TestGraph.h"

// **********TestGraphIterator interface specification**********

// ***custom abbreviations***
//        del for delete
//        ex for exists

// *****interface syntax*****

// ***constants***

// ***types***
typedef enum {ARC,NODE} ArcNodeFlag;

// ***class declarations***
class ArcInfo {
public:
        int src,
        int dst,
        int label,

        operator==(ArcInfo arcInfo) {
                return ((arcInfo.src == src) && (arcInfo.dst == dst)
                        && (arcInfo.label == label)),
        }
        operator!=(ArcInfo arcInfo) {
                return ((arcInfo.src != src) || (arcInfo.dst != dst)
                        || (arcInfo.label != label)),
        }
},

class ArcNode {
public:
        ArcNodeFlag flag,
        int id,

        operator==(ArcNode arcNode) {
```

```
                return ((arcNode flag == flag) && (arcNode id == id));
        }
        operator!=(ArcNode arcNode) {
                return ((arcNode flag != flag) || (arcNode id != id));
        }
},

typedef List<ArcNode> Path,

class TestGraphIterator {
public
        TestGraphIterator(const TestGraph&),
        ~TestGraphIterator(),

        void reset(),
        int isEnd(),
        Path next(),
// ***private information---begin***
private
        const TestGraph& testgraph,
        Path *curPath,
        int first,
        int visited[MAXNODES][MAXNODES], // arcs visited since reset

        int pathIndex, // index of the current node in curPath
        ListSet<Path> *pathSet, // set of paths returned by next()
        ListSetIter<Path> *pathIterator,

        // local functions
        ListSet<ArcInfo>& getOutArcs(int),
        void nextArc(int),
// ***private information---End***
},

// ***concrete state invariant***
//         (forall i,j in [0..testgraph.numNode-1])(visited[i][j] = 1
//           => the arc from testgraph.node[i] to testgraph.node[j] has
//           been traversed since the iterator was last initialized)
//

// ***abstraction function****
```

```
// ***exception handlers***
void TGEndExc(),
void TGNoStartExc();

/*****interface semantics---begin*****

***abstract state variables***
        graph: testgraph
        anSet: set of integer
        pSet: set of path

***abstract state invariant***
        (anSet = graph.arcSet+graph.nodeSet) and
        (forall p in pSet)(a in p => a in anSet)

***graph coverage criteria***
        Every arc reachable from the start node is visited at least once.

***iterator policy***
        The iterator will generate a set of rooted paths guaranteed to
        satisfy the coverage criteria.

***member functions***

TestGraphIterator(tg):
        transition: graph,anSet,pSet = tg,set of arcs and nodes in tg,
                    set of paths in tg generated according to coverage
                    criteria
        exceptions: exc = (not tg.startNode in tg.anSet => TGNoStartExc)
        assumptions: none

reset():
        transition: pSet = set of paths in graph satisfying the
                    coverage criteria
        exceptions: exc = (not graph.startNode in anSet => TGNoStartExc)
        assumptions: none

next():
        transition-output: pSet,out = pSet-{p},p where p is in pre(pSet)
        exceptions: exc = (pSet == {} => TGEndExc)
        assumptions: testgraph is not modified since last reset call
```

```
isEnd()
        output: out = (no unreturned paths in pSet)
        exception: exc = none
        assumptions: none

***local types***
        path: sequence of integer

*****interface semantics---End*****/
```

A.3 TestGraph class implementation

```
#include <iostream.h>
#include "TestGraph.h"

// ***custom abbreviations***
//        del for delete
//        ex for exists

// ***public member functions***

TestGraph::TestGraph(int start)
{
        numNode = 0;
        startNode = start;
}

void TestGraph::setStartNode(int node)
{
        startNode = node;
}

int TestGraph::getStartNode() const
{
        return(startNode);
}

void TestGraph::addNode(int addNode)
{
        if (numNode == MAXNODES) {
                TGFullExc();
```

```
                return;
        }
        if (findPos(addNode) != numNode) {
                TGExNodeExc();
                return;
        }
        // add node
        node[numNode++] = addNode;
        // set arcs to non-existent
        for (int i = 0; i < numNode; i++) {
                arc[i][numNode-1].exist = 0;
                arc[numNode-1][i].exist = 0;
        }
}

void TestGraph::delNode(int deleted)
{
        int index;

        if ((index = findPos(deleted)) == numNode) {
                TGNotExNodeExc();
                return;
        }
        // delete node
        node[index] = node[numNode-1];
        // delete arcs
        for (int j = 0; j < numNode; j++) {
                if (index != j) {
                        arc[index][j].exist = arc[numNode-1][j].exist;
                        arc[index][j].label = arc[numNode-1][j].label;
                        arc[j][index].exist = arc[j][numNode-1].exist;
                        arc[j][index].label = arc[j][numNode-1].label;
                }
        }
        arc[index][index].exist = arc[numNode-1][numNode-1].exist;
        arc[index][index].label = arc[numNode-1][numNode-1].label;
        // check start node
        numNode--;
}

int TestGraph::exNode(int node) const
{
```

```
        return(findPos(node) != numNode),
}

int TestGraph::getNumNode() const
{
        return(numNode),
}

void TestGraph::addArc(int src,int dst,int label)
{
        int srcIndex = 0,dstIndex = 0,

        // check both nodes exist
        if ((srcIndex = findPos(src)) == numNode) {
                TGNotExNodeExc(),
                return,
        }
        if ((dstIndex = findPos(dst)) == numNode) {
                TGNotExNodeExc(),
                return,
        }
        // check if arc already exists
        if (arc[srcIndex][dstIndex].exist) {
                TGExArcExc(),
                return,
        }
        // add arc
        arc[srcIndex][dstIndex].exist = 1,
        arc[srcIndex][dstIndex].label = label,
}

void TestGraph::delArc(int src,int dst)
{
        int srcIndex,dstIndex,

        // check arc existance
        srcIndex = findPos(src),
        dstIndex = findPos(dst),
        if ((srcIndex == numNode) || (dstIndex == numNode) ||
        (!arc[srcIndex][dstIndex].exist)) {
                TGNotExArcExc(),
                return,
```

```
        }
        // delete arc
        arc[srcIndex][dstIndex] exist = 0;
}

int TestGraph::exArc(int src,int dst) const
{
        int srcIndex,dstIndex;

        srcIndex = findPos(src);
        dstIndex = findPos(dst);
        return((srcIndex != numNode) && (dstIndex != numNode) &&
        arc[srcIndex][dstIndex] exist);
}

int TestGraph::getLabel(int src,int dst) const
{
        int srcIndex,dstIndex;

        srcIndex = findPos(src);
        dstIndex = findPos(dst);
        if ((srcIndex == numNode) || (dstIndex == numNode) ||
        (!arc[srcIndex][dstIndex] exist)) {
                TGNotExArcExc();
                return(0);
        }
        return(arc[srcIndex][dstIndex] label);
}

// ***private member functions***

int TestGraph::findPos(int node1) const
{
        int i;

        for (i = 0; i < numNode; i++)
                if (node[i] == node1)
                        return(i);
        return(numNode);
}
```

A.4 TestGraphIterator class implementation

```
#include <iostream.h>
#include "TestGraphIterator.h"

// ***custom abbreviations***
//        del for delete
//        ex for exists

// ***public member functions***

TestGraphIterator::TestGraphIterator(const TestGraph& t1) : testgraph(t1)
{
        first = 1;
        reset();
}

TestGraphIterator::~TestGraphIterator()
{
        if (!first) { // space has been allocated if first is false
                delete curPath;
                delete pathSet;
                delete pathIterator;
        }
}

void TestGraphIterator::reset()
{
        int position;
        if ((position = testgraph.findPos(testgraph.startNode))
                        >= testgraph.numNode) {
                TGNoStartExc();
                return;
        }

        pathIndex = 0;

        if (!first) { // reset was not called from constructor
                delete curPath;
                delete pathSet;
                delete pathIterator;
        }
```

```
        first = 0,
        for (int i = 0, i < testgraph numNode, i++)
                for (int j = 0, j < testgraph numNode, j++)
                                visited[i][j] = 0,

        ArcNode arcNode,
        arcNode flag = NODE,
        arcNode id = testgraph getStartNode(),
        curPath = new Path,
        curPath->addAt(arcNode,pathIndex++),
        pathSet = new ListSet<Path>,
        nextArc(position),
        pathIterator = new ListSetIter<Path>(*pathSet),
}

ListSet<ArcInfo>& TestGraphIterator getOutArcs(int node)
{
        ListSet<ArcInfo> *outArcs = new ListSet<ArcInfo>,

        for (int i = 0, i < testgraph numNode, i++) { // for all nodes
                // if out-arc exists and has not yet been visited
                if (!visited[node][i] && testgraph arc[node][i] exist) {
                        ArcInfo arc,
                        arc src = node,
                        arc dst = i,
                        arc label = testgraph arc[node][i] label,
                        outArcs->add(arc),
                }
        }
        return *outArcs,
}

void TestGraphIterator nextArc(int curNode)
{
        ListSet<ArcInfo>& outArcSet = getOutArcs(curNode),
        ListSetIter<ArcInfo> iterator(outArcSet),

        while (!iterator isEnd()) {
                ArcInfo arc = iterator next(),
                if (!visited[arc src][arc dst]) {
                        visited[arc src][arc dst] = 1,
                        ArcNode arcNode,
```

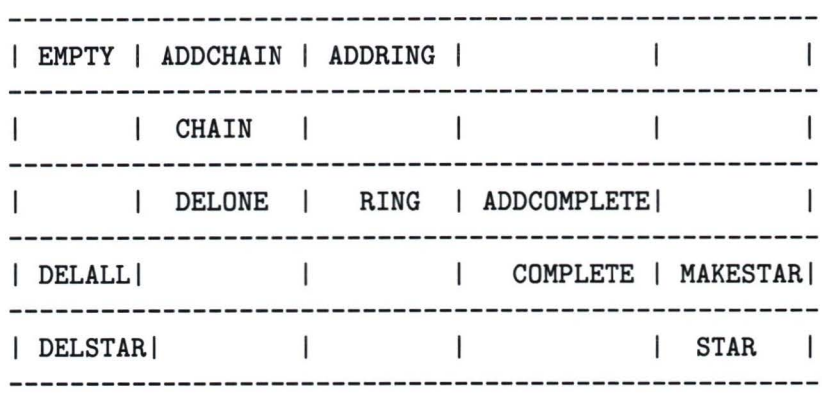
```
                        arcNode flag = ARC,
                        arcNode id = arc label,
                        curPath->addAt(arcNode,pathIndex++),
                        arcNode flag = NODE,
                        arcNode id = testgraph node[arc dst], // node id
                        curPath->addAt(arcNode,pathIndex++),
                        nextArc(arc dst), // recursive call for dst node
                }
        }

        if (outArcSet size() == 0)
                pathSet->add(*curPath),

        if ((curNode == testgraph findPos(testgraph startNode)) &&
            (curPath->size() == 1)) // if startnode only element in path
                curPath->removeAt(--pathIndex), // remove one element
        else { // if (outArcSet size() > 1)
                curPath->removeAt(--pathIndex), // remove last node
                curPath->removeAt(--pathIndex), // remove last arc
        }

        delete &outArcSet,
}

Path TestGraphIterator next()
{
        if (pathIterator->isEnd()) {
                TGEndExc(),
                return (*curPath),
        }
        return pathIterator->next(),
}

int TestGraphIterator isEnd()
{
        return (pathIterator->isEnd()),
}
```

Appendix B

The TestGraph test suite

B.1 TestGraph test plan

```
CUSTOM ABBREVIATIONS
    del: delete

TEST SUITE PARAMETERS
    length: integer in [0,MAXNODES]
    selected values: <0,1,3,10,MAXNODES>

ASSUMPTIONS
    MAXNODES >= 1

TEST CASE SELECTION STRATEGY

Specification-based tests
    Normal case
            Special values for object state
                    interval rule on the size of testgraphs stored
            Special values for member function parameters
                    interval rule on add order for get calls
    Exceptions
            Generate each exception at least once

Program-based tests
    Achieve 100% statement coverage

TESTGRAPH
```




```
Numbering scheme:
    For non STAR graphs
        Nodes are labelled from 0 to length-1, start node is 0.
    For a STAR graph,
        node labels are permutations of all digits from 1 to length,
        start node is 12...length,
        For all i in [2,length], each node s has an arc to and from
        the length-1 other nodes labelled by the numbers obtained by
        interchanging the first and i'th digits of s.
    The arc from node s to node d has the label s+d

ABSTRACTION FUNCTION
    A(node,length) =
        (node = EMPTY => empty TestGraph
        | node = RING  => ring of 'length' nodes, start node is 0
        | node = CHAIN => chain of 'length' nodes, start node is 0
        | node = COMPLETE => complete graph of 'length' nodes, start node 0
        | node = STAR => star graph of arity 'length', start node is 1)

ORACLE STRATEGY

Instance variables
    length: graph parameter (arity for star, number of nodes for others)
    arcs: MAXNODES x MAXNODES adjacency matrix, nodes lie on main
        diagonal, arcs elsewhere, each matrix entry has
                a field for node/arc numbers
                a field showing if the node/arc has been visited.
    node: current driver testgraph node
    numArcs: number of arcs in the CUT
    numNodes: number of nodes in the CUT
```

```
Public member functions
    constructor(int size)
        length,node,numNodes,numArcs = size,EMPTY,0,0
    destructor
    setGraph(int nodeIdentity)
        node = nodeIdentity
        obtain set of nodes, s, using nodeSet
        for each i in s, insert i at arcs[j][j] such that j in [0,|s|-1]
        obtain set of arcs, a, using outArcs
        for each i in a, store i label at arcs[i src][i dst]
    getNumNode()
        return numNodes
    isPath(List<ArcNode>& path)
        checks that path is a valid path in the CUT
        and the ArcNodes refer to valid arcs and nodes in the CUT
        (path is traversed by following the arcs and nodes in order)
    allDone()
        Verifies that each matrix entry having a valid node or arc
        has been covered

Private member functions
    getExistNode(int nodeName)
        if nodeName is a valid node in arcs[], return 1 else return 0
    getLabel(int src,int dst)
        returns label of arc from src to dst  On error return -1
    getNode(int src,int dst)
        returns label of arc from arcs[src][src] to arcs[dst][dst]
    getPos(int nodeId)
        returns position of nodeId in arcs[][]

Pattern functions
    adjacent(int src,int dst,int& label)
        if an arc exists between src and dst by the numbering scheme
                return 1
    nodeSet(int pattern,int size)
        calculate set of nodes by numbering scheme for pattern = STAR,
        using createStar()
        return {0,1  ,size-1} for others
    outArcs(int pattern,int size,ListSet<int>& nodes)
        for all a,b in [0,|nodes|-1],
                if arc from nodes[a] to nodes[b] exists
```

```
                    use adjacent() to obtain arc label,
                return set of <nodes[a],nodes[b],label>
    neighbour(int size,int src,int dst)
        use the definition of star graphs to check if src and dst
        are neighbours in a star graph of size arity

DRIVER STRATEGY

Instance variables
    cut and orc to store current CUT and oracle

Public member functions
    constructor(int size,int level)
        for value of length and msgLevel
    arc(int a)
        orc setGraph used to load the orc
        changeCUT used to duplicate orc in cut
    node()
        Normal case
            checkCut()
                use CHECKVAL on getNumNode() for orc and cut
                calculate start node for STAR and others
                use cut setStartNode() with start node
                use CHECKVAL on cut getStartNode() and start node
                use next() on cut to get path and isPath() to verify
        Exceptions
            setAddNode(int n)
                Generate exceptions for interval rule on value of n
                using CheckExistNod(), for overflow using CheckFull()
            setStartNode(int n)
                Generate exception for n = length (nonexistant node)
                    using CheckNotExistNod(),
            setAddArc(int s,int d,int l)
                Generate exceptions for an illegal src (s=length+1),
                for an illegal dst (d = length+1) and interval rule
                on existing arcs using CheckNotExistNode() and
                CheckExistArc()
            getLabel(int s,int d)
                Generate exception for a nonexistant arc
                (length,length+1) using CheckNotExistArc()
            isEnd()
                Generate exception with no start node specified
```

```
            using CheckNoStart()
        next()
            Generate exception with no start node specified
            using CheckNoStart() and with no more paths using
            CheckEnd()

Private member functions
    changeCUT(srcOrc,dstOrc)
            make calls on cut to change from srcOrc into dstOrc  cut
            should resemble dstOrc after this function call
    CheckExistNodExc(TestGraph& c,Oracle& o)
            check exception exNod occurs for c setAddNode(n) with
            n = 0, o length-1, and an intermediate value
    CheckExistArcExc(TestGraph& c,Oracle& o)
            check exception exArc occurs for c setAddArc(s,d,a) with s,d,a
            = 0,1,1, and o length-1,0,o length-1, and an intermediate value
    CheckFullExc(TestGraph& c,Oracle& o)
            if o length = MAXNODES check exception full in c setAddNode
    CheckNotExistNodExc(TestGraph& c,Oracle& o)
            check exception notExistNod in c setStartNode(n) for n = length
            c setAddArc(0,o length,?), and c setAddArc(o length,0,?)
    CheckNotExistArcExc(TestGraph& c,Oracle& o)
            check exception notExistArc in c getLabel(o length,o length+1)
            and on c delArc(o length,o length+1)
    CheckNoStartExc(TestGraph& c,TGIterator& i,Oracle& o)
            add a node by c setAddNode(o length),
            set the start node by c setStarNode(o length),
            delete the node with c setDelNode(o length),
            check that exception noStart occurs in i next()
    CheckEndExc(TestGraph& c,TGIterator& i,Oracle& o)
            check exception end occurs by calling i next() when
            o node is in state RING, and i isEnd() returns non-zero
```

B.2 Oracle Class Interface Specifications

```
#include "TestGraphIterator.h"

// **********Oracle Interface Specification**********

// *****interface syntax*****
```

```
// ***constants***

// ***types***

struct NodeType {
        int value,
        int visited,
},

// ***class declaration***
class Oracle {
public
        Oracle(int),
        ~Oracle(),

        void setGraph(int),
        int getNumNode() const,
        int getLabel(int,int) const, // expects node labels
        int getNode(int,int) const, // expects node/arc indices
        int isPath(Path*),
        int allDone() const,

        // node identifier
        int node,

        // graph parameter
        int length,

// ***private information---begin***
private
        // other instance variables
        NodeType arcs[MAXNODES][MAXNODES], // stores nodes and arcs

        int getExistNode(int) const,
        int getPos(int) const,
        int numArc,
        int numNode,
},

// ***concrete state invariant***
//      node in {EMPTY,RING,CHAIN,COMPLETE,STAR} and
//      length in [0,MAXNODES]
```

```
// ***abstraction function***
//      See TestPlan

// ***private information---end***

// ***public pattern functions***
ListSet<int>& nodeSet(int,int); // caller frees up the memory
ListSet<ArcInfo>& arcSet(int,int,ListSet<int>&); // caller frees memory
int adjacent(int,int,int,int);
int flip(int,int, int);

// ***exception handlers***

/*****interface semantics---begin*****

***abstract state variables***
nodeSet: set of integers
arcSet: set of integers

***member functions***
Oracle(n):
        transition: length,node,arcSet,nodeSet  = n,EMPTY,{},{}
        exceptions: none
        assumptions: none

~Oracle():
        transition: none
        exceptions: none
        assumptions: none

setGraph(n):
        transition: arcs  = (n = EMPTY => arcSet = {}
                | n = RING or n = CHAIN or n = COMPLETE =>
                        (forall i in [0,length-1])(nodeSet += i),
                        arcSet  = set of arcs for pattern n
                        set of arcs for pattern n in arcs
                | n = STAR => (n < 1 or n > 5 => arcs = {}
                         | set of star nodes in nodeSet
                           set of star arcs in arcSet))
        exceptions: none
        assumptions: none
```

```
getNumNode()
        output: out := (node == EMPTY => 0 | true => length)
        exceptions: none
        assumptions: none

getLabel(i,j)
        output: out := (i != j => (exists an arc from i to j in arcSet
                                   => label of arc from i to j in arcSet
                                   | true => -1)
                        | (node i exists in nodeSet => i | true => -1))
        exceptions: none
        assumptions: none

getNode(i,j)
        output: out := (i != j => (exists an arc from positions i to j in
                                arcSet => label of that arc | true => -1)
                        | (exists node at i in nodeSet => i | true => -1))
        exceptions: none
        assumptions: none

isPath(p)
        output: out := (p is a valid path from arcs in arcSet)
        exceptions: none
        assumptions: none

allDone()
        output: out := (all nodes and arcs in nodeSet and arcSet visited)
        exceptions: none
        assumptions: none

*****interface semantics---end*****/
```

B.3 Oracle class implementation

```
#include <iostream.h>
#include "label.h"
#include "Oracle.h"

Oracle::Oracle(int size)
{
        length = size;
```

```
        node = EMPTY;
        numNode = numArc = 0;
        // set all arcs and nodes to unvisited
        for (int i = 0; i < MAXNODES; i++) {
                for (int j = 0; j < MAXNODES; j++) {
                        arcs[i][j].value = -1;
                        arcs[i][j].visited = -1;
                }
        }
}

Oracle::~Oracle()
{
        // intentionally empty
}

void Oracle::setGraph(int newNode)
{
        for (int i = 0,j; i < MAXNODES; i++) { // mark arcs unvisited
                for (j = 0; j < MAXNODES; j++) {
                        arcs[i][j].value = -1;
                        arcs[i][j].visited = -1;
                }
        }
        node = newNode;
        ListSet<int>& setOfNodes = nodeSet(node,length);
        ListSetIter<int> nodeIter(setOfNodes);
        for (numNode = 0, i = 0; i < setOfNodes.size(); i++) {
                arcs[i][i].value = nodeIter.next(); // nodes on diagonal
                numNode++;
        }
        ListSet<ArcInfo>& setOfArcs = arcSet(node,length,setOfNodes);
        ListSetIter<ArcInfo> arcIter(setOfArcs);
        for (numArc = 0,i = 0; i < setOfArcs.size(); i++) {
                ArcInfo a = arcIter.next();
                arcs[getPos(a.src)][getPos(a.dst)].value = a.label;
                numArc++;
        }
        delete &setOfNodes;
        delete &setOfArcs;
}
```

```
int Oracle::getPos(int nodeId) const
{
        for (int i = 0; i < MAXNODES; i++) {
                if (arcs[i][i].value == nodeId)
                        return i;
        }
        return -1;
}


int Oracle::getNumNode() const
{
        return numNode;
}

int Oracle::isPath(Path *path)
{
        int previous,arc,pathLength = path->size();

        if (pathLength == 0) // empty path always in testgraph
                return 1;

        if ((length == 0) && (pathLength == 0))
                return 1;
        ArcNode arcNode = (*path)[0];
        previous = arcNode.id;
        if (node != STAR) {
                if (previous != 0)
                        return 0;  // incorrect start node
        }
        if (!getExistNode(previous)) // test for STAR's start node
                return 0;
        arcs[getPos(previous)][getPos(previous)].visited = 1;
        // test the remaining path
        for (int i = 1; i < pathLength; i++){
                arcNode = (*path)[i];
                if (arcNode.flag == ARC)
                        arc = arcNode.id;
                else {
                        if (!getExistNode(arcNode.id))
                                return 0;
                        if (previous != arcNode.id) { // no self loops
```

```
                                if (arc != getLabel(previous,arcNode id))
                                        return 0,
                                arcs[getPos(previous)][getPos(arcNode
                                id)] visited = 1, // arc visited
                                arcs[getPos(arcNode id)][getPos(arcNode
                                id)] visited = 1, // node visited
                        }
                        previous = arcNode id,
                }
        }
        return 1,
}

int Oracle getExistNode(int nodeId) const
{
        int lower, upper,index,

        upper = numNode,
        lower = 0,
        index = getPos(nodeId),
        if (index < lower || index >= upper)
                return 0,
        return 1, // if index is within limits, the node exists
}

// this function expects node/arc indices as parameters
// and returns the label at that position in arcs[][]
int Oracle getNode(int src,int dst) const
{
        if (src < 0 || src > numNode)
                return -1,
        if (dst < 0 || dst > numNode)
                return -1,
        return (arcs[src][dst] value),
}

// this function expects node labels as parameters
// and returns the arc label between them in arcs[][]
int Oracle getLabel(int src,int dst) const
{
        src = getPos(src), // obtain node index from label
        dst = getPos(dst),
```

```
        return getNode(src,dst);
}

int Oracle::allDone() const
{
        if (length == 0)
                return 1;
        for (int i = 0; i < numNode; i++) // for all entries
                for (int j = 0; j < numNode; j++)
                        if (arcs[i][j].value != -1) // if exists
                                if (arcs[i][j].visited != 1) // unvisited
                                        return 0;
        return 1;
}
```

B.4 Driver Class Interface Specifications

```
#include "TestGraphIterator.h"
#include "AbstractDriver.h"
#include "Oracle.h"

// **********Class Interface Specification**********

// *****interface syntax*****

// ***constants***
const int FIVE = 5; // used for star

// ***types***

// ***class declaration***
class Driver : public AbstractDriver {
public:
        Driver(int);
        void reset();
        void arc(int);
        void node();

// ***private information---begin***
private:
```

```
        TestGraph cut;
        Oracle orc,

        // cut modifiers
        void changeCut(const Oracle&,const Oracle&),
        void makeNodes(const Oracle&,const Oracle&,int),
        void makeArcs(const Oracle&,const Oracle&,int),
        void callAddArc(int,int),
        void callDelArc(int,int),

        // cut state checking functions
        void checkCut(TestGraph&,Oracle&),
        void checkGraph(TestGraph&,Oracle&) const,
        void checkStartNode(TestGraph&,Oracle&),

        // exception checking
        void checkExistNodeExc(TestGraph&,Oracle&),
        void checkExistArcExc(TestGraph&,Oracle&),
        void checkFullExc(TestGraph&,Oracle&),
        void checkNotExistNodeExc(TestGraph&,Oracle&),
        void checkNotExistArcExc(TestGraph&,Oracle&) const,
        void checkNoStartExc(TestGraph&,Oracle&),
        void checkEndExc(TestGraph&,TestGraphIterator&,Oracle&) const,
};

// ***concrete state invariant***
//      cut and orc are in same abstract state
//
// ***abstraction function***
//      none

// ***private information---end***

// ***exception handlers***

void exArcExc(),
void exNodeExc(),
void fullExc(),
void noStartExc(),
void notExArcExc(),
void notExNodeExc(),
void endExc(),
```

```
/*****interface semantics---begin*****

Omitted same as for AbstractDriver

*****interface semantics---end*****/
```

B.5 Driver class implementation

```
#include <iostream.h>
#include <string.h>
#include "utility.h"
#include "label.h"
#include "Driver.h"

// ***custom abbreviations***
//        del for delete
//        ex for exists

// ***exception code***
enum {NOEXC = -1,EXISTARCEXC,EXISTNODEEXC,FULLEXC,NOSTARTEXC,
NOTEXISTARCEXC,NOTEXISTNODEEXC,ENDEXC,NUMEXC},

ExcTable excTable(NUMEXC),

// ***public member functions***

Driver::Driver(int size) : orc(size)
{
        // intentionally empty
}

void Driver::reset()
{
        Oracle *orc1 = new Oracle(orc.length),
        changeCut(orc,*orc1),
        delete orc1,
        orc.setGraph(EMPTY),
}

void Driver::arc(int arc)
```

```
{
        Oracle *orc1;

        switch (arc) {
        case ADDRING:
                orc1 = new Oracle(orc.length);
                orc.setGraph(RING);
                changeCut(*orc1,orc);
                break;
        case ADDCHAIN:
                orc1 = new Oracle(orc.length);
                orc.setGraph(CHAIN);
                changeCut(*orc1,orc);
                break;
        case DELONE:
                orc1 = new Oracle(orc.length);
                orc1->setGraph(RING);
                orc.setGraph(CHAIN);
                changeCut(*orc1,orc);
                break;
        case ADDCOMPLETE:
                orc1 = new Oracle(orc.length);
                orc1->setGraph(RING);
                orc.setGraph(COMPLETE);
                changeCut(*orc1,orc);
                break;
        case DELALL:
                orc1 = new Oracle(orc.length);
                orc1->setGraph(COMPLETE);
                orc.setGraph(EMPTY);
                changeCut(*orc1,orc);
                break;
        case MAKESTAR:
                orc1 = new Oracle(orc.length);
                orc1->setGraph(COMPLETE);
                orc.setGraph(STAR);
                changeCut(*orc1,orc);
                break;
        case DELSTAR:
                orc1 = new Oracle(orc.length);
                orc1->setGraph(STAR);
                orc.setGraph(EMPTY);
```

```
                changeCut(*orc1,orc);
                break;
        }
        delete orc1;
}

void Driver::node()
{
        // normal case
        checkCut(cut,orc);

        // exceptions
        excTable::checkExc();
        checkExistNodeExc(cut,orc);
        checkExistArcExc(cut,orc);
        checkFullExc(cut,orc);
        checkNotExistNodeExc(cut,orc);
        checkNotExistArcExc(cut,orc);
}


// ***private member functions***

void Driver::changeCut(const Oracle& src,const Oracle& dst)
{
        PRINTLOGMSG1(1,"changeCut called");

        int srcLength = src.getNumNode(),dstLength = dst.getNumNode();
        int tLength = ((srcLength < dstLength) ? dstLength : srcLength);

        makeNodes(src,dst,tLength);
        makeArcs(src,dst,tLength);
}

// make sure the same nodes are present in CUT and orc
void Driver::makeNodes(const Oracle& src,const Oracle& dst,int length)
{
        for (int s = 0; s < length; s++) {
                if (src.getNode(s,s) != dst.getNode(s,s)) {
                    if ((src.getNode(s,s) == -1) && (dst.getNode(s,s)
                        != -1)) { // node in dst, but not in src
                        cut.addNode(dst.getNode(s,s));
```

```
                    PRINTLOGMSG2(2,"\tAdd node  ",dst getNode(s,s));
                } else if ((dst getNode(s,s) == -1) &&
                                (src getNode(s,s) != -1)) {
                    // node in src, but not in dst
                    cut delNode(src getNode(s,s)); // removes arcs
                    PRINTLOGMSG2(2,"\tDel node  ",src getNode(s,s));
                } else {
                    // different nodes in src and dst
                    cut delNode(src getNode(s,s));
                    PRINTLOGMSG2(2,"\tDel node  ",src getNode(s,s));
                    cut addNode(dst getNode(s,s));
                    PRINTLOGMSG2(2,"\tAdd node  ",dst getNode(s,s));
                }
            }
        }
}

// produces the same arcs in CUT as in orc; arc deletion needs to occur
// only if the node ids are same for src and dst, but with different
// arc labels, or arcs are removed, but not their nodes
void Driver::makeArcs(const Oracle& src,const Oracle& dst,int length)
{
        for (int s = 0; s < length; s++) {
                for (int d = 0; d < length; d++) {
                        if (s != d) {
                                if ((src getNode(s,d) != -1) && (dst
                                getNode(s,d) != -1)) { // different arcs
                                        if ((src getNode(s,s) ==
                                        dst getNode(s,s)) && (src getNode
                                        (d,d) == dst getNode(d,d)))
                                        // arcs with the same s,d nodes
                                                callDelArc(src getNode
                                                (s,s),src getNode(d,d));
                                        callAddArc(dst getNode(s,s),
                                        dst getNode(d,d));
                                } else if ((src getNode(s,d) != -1) &&
                                (dst getNode(s,d) == -1) &&
                                (src getNode(s,s) == dst getNode(s,s)) &&
                                (src getNode(d,d) == dst getNode(d,d))) {
                                        // src,dst have same node
                                        callDelArc(src getNode(s,s),
                                        src getNode(d,d)); // remove arc
```

```
                                } else if ((src getNode(s,d) == -1) &&
                                (dst getNode(s,d) != -1)) { // arc in dst
                                        callAddArc(dst getNode(s,s),
                                        dst getNode(d,d)),
                                }
                        }
                }
        }
}

void Driver callAddArc(int src,int dst)
{
        PRINTLOGMSG2(3,"\tAdd Arc from  ",src),
        PRINTLOGMSG2(3,"\tAdd Arc to    ",dst),
        PRINTLOGMSG2(3,"\tAdd Arc label ",src+dst),
        cut addArc(src,dst,src+dst),
}

void Driver callDelArc(int src,int dst)
{
        PRINTLOGMSG2(3,"\tDel Arc from  ",src),
        PRINTLOGMSG2(3,"\tDel Arc To    ",dst),
        PRINTLOGMSG2(3,"\tDel Arc label ",src+dst),
        cut delArc(src,dst),
}

void Driver checkCut(TestGraph& cut1, Oracle& orc1)
{
        if (orc1 length != 1)
                CHECKVAL(orc1 getNumNode(),cut1 getNumNode()),

        // check the nodes and arcs in the cut
        checkGraph(cut1,orc1),

        // traverse graph and check path with isPath and allDone
        if (orc1 node != EMPTY) {
                excTable checkExc(),
                checkNoStartExc(cut1,orc1), // before start node is set
                checkStartNode(cut1,orc1), // set start node & exception
                if ((orc1 length > 0) && ((orc1 node != STAR) ||
                ((orc1 node == STAR) && (orc1 length < FIVE)))) {
                        TestGraphIterator iterator(cut1),it2(cut1),
```

```
                        excTable checkExc(),
                        while (!iterator isEnd()) {
                                PRINTLOGMSG1(1,"Checking path"),
                                CHECKVALBOOLEAN(0,orc1 allDone()),
                                Path& p=iterator next(),&p2=it2 next(),
                                CHECKVALBOOLEAN(1,orc1 isPath(&p)),
                                CHECKVALBOOLEAN(1,orc1 isPath(&p2)),
                        }
                        CHECKVALBOOLEAN(1,orc1 allDone()),
                        excTable checkExc(),
                        checkEndExc(cut1,iterator,orc1),
                        checkEndExc(cut1,it2,orc1),
                }
        }
}

void Driver checkGraph(TestGraph& cut1,Oracle& orc1) const
{
        int label,num = orc1 getNumNode(),

        excTable checkExc(),
        for (int i = 0, i < num, i++) {
                if ((label = orc1 getNode(i,i)) != -1)
                        CHECKVAL(1,cut1 exNode(label)),
                for (int j = 0, j < num, j++) {
                        if ((i != j) && ((label = orc1 getNode(i,j))
                        != -1)) {
                                int s = orc1 getNode(i,i),
                                int d = orc1 getNode(j,j),
                                CHECKVAL(1,cut1 exArc(s,d)),
                                CHECKVAL(label,cut1 getLabel(s,d)),
                        }
                }
        }
}

void Driver checkStartNode(TestGraph& cut1,Oracle& orc1)
{
        TestGraph cut2,
        int i,j,number = 0,

        CHECKVAL(0,cut2 getStartNode()),
```

```
        if ((orc1 node == STAR) && (orc1 length > 0)) {
                for (i = orc1 length,j = 1, i >= 1, i--,j *= 10)
                        number += i*j,
        }
        cut1 setStartNode(number),
        CHECKVAL(number,cut1 getStartNode()),
}

// exception testing

void Driver checkExistNodeExc(TestGraph& cut1,Oracle& orc1)
{
        int nodes = orc1 getNumNode(),
        if (nodes != MAXNODES && nodes != 0) { // avoid fullExc
                PRINTLOGMSG1(1,"Checking exception ExistNode"),
                if (orc1 node != STAR)
                    for (int i = 0; i < 3; i++){
                        excTable clearExc(),
                        excTable setExpFlag(EXISTNODEEXC,1),
                        switch (i) {
                                case 0: cut1 addNode(0),
                                        break,
                                case 1: cut1 addNode(orc1 length-1),
                                        break,
                                case 2: cut1 addNode(orc1 length/2),
                                        break,
                        }
                        excTable checkExc(),
                    }
                else if ((orc1 length > 0) && (orc1 length < FIVE)) {
                        for (int i = orc1 length,j = 1,number = 0,
                        i >= 1, i--,j *= 10)
                                number += i*j,
                        excTable clearExc(),
                        excTable setExpFlag(EXISTNODEEXC,1),
                        cut1 addNode(number),
                        excTable checkExc(),
                        if (orc1 length > 1) { // flip 1st and last
                                int number2 = flip(orc1 length,1,number),
                                excTable clearExc(),
                                excTable setExpFlag(EXISTNODEEXC,1),
```

```
                                cut1 addNode(number2),
                                excTable checkExc(),
                        }
                }
                excTable clearExc(),
        }
}

void Driver checkExistArcExc(TestGraph& cut1,Oracle& orc1)
{
        PRINTLOGMSG1(1,"Checking exception ExistArc"),
        excTable clearExc(),
        for(int i = 0,s,d, i < orc1 getNumNode(), i++) {
                s = orc1 getNode(i,i),
                for(int j = 0, j < orc1 getNumNode(), j++) {
                        d = orc1 getNode(j,j),
                        // for all neighbours, add the arc, test exc
                        if (adjacent(orc1 node,orc1 length,s,d)) {
                                excTable setExpFlag(EXISTARCEXC,1),
                                cut addArc(s,d,s+d),
                                excTable checkExc(),
                                excTable clearExc(),
                        }
                }
        }
}

void Driver checkFullExc(TestGraph& cut1,Oracle& orc1)
{
        PRINTLOGMSG1(1,"Checking exception Full"),
        if ((orc1 length == MAXNODES) && (orc1 node != EMPTY))
                if ((orc1 node == STAR) && // STAR => length < 5 or > 0
                ((orc1 length >= FIVE) || (orc1 length < 1)))
                        return,
                else {
                        excTable clearExc(),
                        excTable setExpFlag(FULLEXC,1),
                        cut1 addNode(MAXNODES), // adding 1 signals full
                        excTable checkExc(),
                        excTable clearExc(),
                }
}
```

```
void Driver::checkNotExistNodeExc(TestGraph& cut1,Oracle& orc1)
{
        PRINTLOGMSG1(1,"Checking exception NotExistNode");
        excTable.clearExc();
        excTable.setExpFlag(NOTEXISTNODEEXC,1);
        cut1.delNode(cut1.getNumNode()+1);
        excTable.checkExc();
        excTable.clearExc();
        if ((orc1.node == STAR) && (orc1.length == 0))
                return; // arc<0,0,0> allowed
        else {
                excTable.setExpFlag(NOTEXISTNODEEXC,1);
                cut1.addArc(orc1.length,0,0); // node 'length' not added
                excTable.checkExc();
                excTable.clearExc();
                excTable.setExpFlag(NOTEXISTNODEEXC,1);
                cut1.addArc(0,orc1.length,0);
                excTable.checkExc();
                excTable.clearExc();
        }
}

void Driver::checkNotExistArcExc(TestGraph& cut1,Oracle& orc1) const
{
        PRINTLOGMSG1(1,"Checking exception NotExistArc");
        excTable.clearExc();
        excTable.setExpFlag(NOTEXISTARCEXC,1);
        cut1.getLabel(orc1.length,orc1.length+1);
        excTable.checkExc();
        excTable.clearExc();
        excTable.setExpFlag(NOTEXISTARCEXC,1);
        cut1.delArc(orc1.length,orc1.length+1);
        excTable.checkExc();
        excTable.clearExc();
}

void Driver::checkNoStartExc(TestGraph& cut1,Oracle& orc1)
{
        PRINTLOGMSG1(1,"Checking exception NoStart");
        if (orc1.length < MAXNODES) {
                cut1.addNode(-1); // this node never in any graph
```

```
                int startNode = cut1.getStartNode();
                cut1.setStartNode(-1);
                TestGraphIterator test1(cut1);
                cut1.delNode(-1); // leaves start node invalid
                excTable.checkExc();
                excTable.setExpFlag(NOSTARTEXC,1);
                TestGraphIterator test2(cut1); // raises noStartExc
                excTable.checkExc();
                excTable.clearExc();
                excTable.setExpFlag(NOSTARTEXC,1);
                test1.reset(); // raises noStartExc
                excTable.checkExc();
                excTable.clearExc();
                cut1.setStartNode(startNode);
        }
}

void Driver::checkEndExc(TestGraph& cut1,TestGraphIterator& i,
Oracle& orc1) const
{
        PRINTLOGMSG1(1,"Checking exception EndExc");
        if (i.isEnd()) {
                excTable.clearExc();
                excTable.setExpFlag(ENDEXC,1);
                i.next();
                excTable.checkExc();
                excTable.clearExc();
        }
}

// exception handlers

void TGExArcExc()
{
        excTable.setActFlag(EXISTARCEXC,1);
}

void TGExNodeExc()
{
        excTable.setActFlag(EXISTNODEEXC,1);
}
```

```
void TGFullExc()
{
        excTable.setActFlag(FULLEXC,1);
}

void TGNoStartExc()
{
        excTable.setActFlag(NOSTARTEXC,1);
}

void TGNotExArcExc()
{
        excTable.setActFlag(NOTEXISTARCEXC,1);
}

void TGNotExNodeExc()
{
        excTable.setActFlag(NOTEXISTNODEEXC,1);
}

void TGEndExc()
{
        excTable.setActFlag(ENDEXC,1);
}

// exception handlers for utility code
void duplicateExc()
{
        cout << "Exception duplicateExc called " << endl;
}

void newFailExc()
{
        cout << "Exception newFailExc called " << endl;
}

void notFoundExc()
{
        cout << "Exception notFoundExc called " << endl;
}

void iterEndExc()
```

```
{
        cout << "Exception iterEndExc called " << endl;
}

void rangeExc()
{
        cout << "Exception rangeExc called " << endl;
}
```

B.6 The pattern routine implementations

```
#include "pattern.h"

ListSet<int>& nodeSet(int pattern,int size)
{
        ListSet<int> *set = new ListSet<int>;
        switch(pattern) {
            case EMPTY:
                    return *set;
            case RING:
            case CHAIN:
            case COMPLETE:
                for (int i = 0,j = 0; i < size; i++)
                        set->add(i);
                return *set;
            case STAR:
                starCount = 0;
                numStarNodes = 1;
                for (i = 0; (i < size) ; i++) {
                        // calculate factorial
                        numStarNodes = numStarNodes * (i+1) ;
                        if (numStarNodes >= INT_MAX) {
                                numStarNodes = 0;
                                return *set;
                        }
                }
                if ((numStarNodes > MAXNODES) || (numStarNodes == 0)) {
                        // cannot compare long numStarNodes with < 0
                        return *set;
                }
                for (i = 0; i < MAXNODES; i++)
```

```
                map[i] = -1;
        for (map[0] = 0,powerOfTen = 1,i = size,j = 1,
                i >= 1, i--, j *= 10) {
                map[0] += i*j;
                powerOfTen *= 10;
        }
        createStar(size,map[0]);
        if (size == 0)
                numStarNodes = 0; // so that none are added
        for (i = 0; i < numStarNodes; i++)
                set->add(map[i]);
        return *set;
    }
}

ListSet<ArcInfo>& arcSet(int pattern,int size,ListSet<int>& nodes)
{
    int array[MAXNODES+1];
    ListSetIter<int> nodeIterator(nodes);
    ListSet<ArcInfo> *arcSet1 = new ListSet<ArcInfo>;

    for (int i = 0; i < nodes.size(); i++)
        array[i] = nodeIterator.next();

    for (i = 0; i < nodes.size(); i++)
        for (int j = 0, label; j < nodes.size(); j++)
            if (array[i] != array[j])
                if (adjacent(pattern,size,array[i],
                array[j])) {
                    ArcInfo a;
                    a.src = array[i];
                    a.dst = array[j];
                    a.label = a.src+a.dst;
                    arcSet1->add(a);
                }
    return *arcSet1;
}

int adjacent(int pattern,int size,int src,int dst)
{
    switch(pattern) {
        case RING:
```

```
                if (dst == (src+1))
                        return 1;
                else if ((src == (size-1)) && (dst == 0) &&
                (dst != src))
                        return 1;
                break;
            case CHAIN:
                if (dst == (src+1))
                        return 1;
                break;
            case COMPLETE:
                if (dst != src)
                        return 1;
                break;
            case STAR:
                if (neighbour(size,src,dst))
                        return 1;
        }
        return 0;
}

void createStar(int size,int num)
{
        int result;

        for (int i = size-1; i >= 1; i--) {
                if (starCount == numStarNodes)
                        return;
                result = flip(size,i,num);
                if (check(result))
                        continue;
                else
                        map[++starCount] = result;
                createStar(size,map[starCount]);
        }
}

int flip(int size,int by, int num)
{
        int digit[10],temp1 = num, power = powerOfTen/10, number, j;

        for (int i = size; i >= 1; i--, power /= 10){
```

```
                digit[i] = temp1 / power;
                temp1 = temp1 % power;
        }
        int temp2 = digit[size];
        digit[size] = digit[by];
        digit[by] = temp2;
        for (i = 1, j = 1, number = 0; i <= size; i++, j *= 10) {
                number += digit[i] * j;
        }
        return number;
}

int check(int result)
{
        for (int i = 0; i < starCount; i++)
                if (map[i] == result)
                        return 1;
        return 0;
}

int neighbour(int size,int src,int dst)
{
        int digit1[10],digit2[10],power = powerOfTen/10;
        int first1,first2,second1,second2,numDifferent, common;

        // store each digit for src in digit1, for dst in digit2
        // highest power of 10 at position size-1, lowest at 0
        for (int i = size-1; i >= 0; i--, power/=10) {
                digit1[i] = src/power;
                src = src%power;
                digit2[i] = dst/power;
                dst = dst%power;
        }

        if (digit1[size-1] == digit2[size-1]) // highest power is same
                return 0; // can't be neighbours

        for (i = 0, numDifferent = 0, common = 0; i < size; i++) {
                if (digit1[i] == digit2[i]) // count common digits
                        common++;
                else  if (numDifferent < 1) { // store 1st different num
                        first1 = digit1[i];
```

```
                        first2 = digit2[i],
                        numDifferent++,
                } else { // stores the next different numbers
                        second1 = digit1[i],
                        second2 = digit2[i],
                        numDifferent++,
                }
        }

        // src and dst are neighbours only if
        // at least size-2 digits are common
        // only two digits are different
        // 1st different digit of src = 2nd different digit of dst
        // 2nd different digit of src = 1st different digit of dst
        if ((common != (size-2)) || (numDifferent != 2) ||
        (first1 != second2) || (second1 != first2))
                return 0,
        return 1,
}
int starToRing(int starNum)
{
        for (int i = 0, i < numStarNodes, i++)
                if (map[i] == starNum)
                        return i,
        return 0,
}
```

Appendix C

The LEDA GRAPH test suite

C.1 Oracle Class Interface Specifications

```
#include "TestGraphIterator.h"

// **********Oracle Interface Specification**********

// *****interface syntax*****

// ***constants***
const int DEPTH = 2, // level of multigraphs

// ***types***

struct NodeType {
        int value,
        int visited[DEPTH],
},

// ***class declaration***
class Oracle {
public:
        friend class Driver,
        Oracle(int),
        ~Oracle(),

        void setGraph(int),
        int getNumNode() const,
        int getNumArc() const,
```

```
        int getExistNode(int) const,
        int getLabel(int,int,int) const,
        int getNode(int,int,int) const,
        int allNodesDone() const,
        int getPos(int) const,
        int isEdge(int,int,int) const,
        int isNode(int) const,

        // node identifier
        int nodeName,

        // graph parameter
        int length,

// ***private information---begin***
private
        // other instance variables
        NodeType arcs[MAXNODES][MAXNODES],

        int numArc,
        int numNode,
},

// ***concrete state invariant***
//      node in {EMPTY,RING,CHAIN}  and length in [0,MAXNODES]
// ***abstraction function***
//      See TP

// ***private information---end***

// ***public pattern functions***
ListSet<int>& nodeSet(int,int), // caller frees the memory
ListSet<ArcInfo>& arcSet(int,int,ListSet<int>&), // caller frees memory
int adjacent(int,int,int,int),
int flip(int,int, int),

// ***exception handlers***

/*****interface semantics---begin*****

***abstract state variables***
nodeSet  set of integers
```

```
arcSet: set of integers

***member functions***
Oracle(n):
        transition: length,node,arcSet,nodeSet := n,EMPTY,{},{}
        exceptions: none
        assumptions: none

~Oracle():
        transition: none
        exceptions: none
        assumptions: none

setGraph(n):
        transition: arcs := (n = EMPTY => arcSet = {}
                | n = RING or n = CHAIN or n = COMPLETE =>
                        (forall i in [0,length-1])(nodeSet += i),
                        arcSet := set of arcs for pattern n
                | n = STAR => (n < 1 or n > 5 => arcs = {}
                        | set of star nodes in nodeSet
                          set of star arcs in arcSet))
        exceptions: none
        assumptions: none

getNumNode():
        output: out := |nodeSet|
        exceptions: none
        assumptions: none

getNumArc():
        output: out := |arcSet|
        exceptions: none
        assumptions: none

getExistNode(n):
        output: out := (exists n in nodeSet)
        exceptions: none
        assumptions: none

getLabel(i,j,k):
        output: out := (i != j => (exists an arc from i to j in arcSet
                        => label of k'th arc from i to j in arcSet
```

```
                        | true => -1)
                    | (node i exists in nodeSet => i | true => -1))
        exceptions: none
        assumptions: none

getNode(i,j,k):
        output: out := (i != j => (exists an arc from node at positions
                                   i to j in arcSet => label of k'th such
                                   arc | true => -1)
                    | (exists node at i in nodeSet => i | true => -1))
        exceptions: none
        assumptions: none

allNodesDone(i):
        output: out := (all nodes in nodeSet visited)
        exceptions: none
        assumptions: none

getPos(i):
        output: out := (exists node i in nodeSet => position of i in
                        nodeSet | true => -1)
        exceptions: none
        assumptions: none

isPath(p):
        output: out := (p is a valid path from arcs in arcSet)
        exceptions: none
        assumptions: none

isEdge(s,d,l):
        output: out := (exists an arc l in arcSet from node s in
                        nodeSet to node d in nodeSet)
        exceptions: none
        assumptions: none

isNode(n):
        output: out := (exists a node n in nodeSet)
        exceptions: none
        assumptions: none

*****interface semantics---end*****/
```

C.2 Driver Class Interface Specifications

```
#include "TestGraphIterator.h"
#include "AbstractDriver.h"
#include "Oracle.h"
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>

// **********Class Interface Specification**********

// *****interface syntax*****

// ***constants***
const int FIVE = 5; // used for star graph limits

// ***types***

// ***class declaration***
class Driver : public AbstractDriver {
public:
        Driver(int),
        void reset(),
        void arc(int),
        void node(),

// ***private information---begin***
private:
        GRAPH<int,int> cut,
        Oracle orc,

        // Cut modifier functions
        void changeCut(const Oracle&,const Oracle&,GRAPH<int,int>&),
        void makeNodes(const Oracle&,const Oracle&,int,GRAPH<int,int>&),
        void makeArcs(const Oracle&,const Oracle&,int,GRAPH<int,int>&),
        void callAddArc(int,int,int,GRAPH<int,int>&),
        void callDelArc(int,int,int,GRAPH<int,int>&),

        // cut check routines
        void checkDegree(GRAPH<int,int>&,Oracle&),
        void checkIterators(GRAPH<int,int>&,Oracle&),
        void checkAll(GRAPH<int,int>&,Oracle&),
        void checkLists(GRAPH<int,int>&,Oracle&),
```

```
        void checkInOut(GRAPH<int,int>&,Oracle&),
        void checkUpdate(GRAPH<int,int>&,Oracle&),
        void checkDirected(GRAPH<int,int>&,Oracle&),
        void checkPaths(GRAPH<int,int>&,Oracle&),

        // local functions
        void checkAdjacent(int *,int *,int,int,node),
        void checkIn(int *,int,node),
        int checkTraversal(list<node>&,Oracle&),

        // local variables
        int sumNodes,
        int sumEdges,
},

// ***concrete state invariant***
//      cut and orc are in same abstract state
//
// ***abstraction function***
//      none

// ***private information---end***

// ***exception handlers***

/*****interface semantics---begin*****

Omitted  same as for AbstractDriver

*****interface semantics---end*****/
```

VITA

Surname: Kumar — Given Names: Kshitij

Place of Birth: DehraDun, India

Educational Institutions Attended:

University of Victoria	1993 to 1995
Maulana Azad College of Technology, Bhopal, India	1986 to 1991

Degrees Awarded:

B.Tech. Maulana Azad College of Technology, Bhopal, India	1991

Honours and Awards:

University Gold Medal, Maulana Azad College of Technology	1991

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis

Testing Graph Classes

Author [redacted]
Kshitij Kumar

24th August 1995
Date