The Testing of Object-Oriented Programs

C. D. Turner and D. J. Robson

Technical Report: TR-13/92

Computer Science Division
School of Engineering and
Computer Science (SECS)
University of Durham
Durham, England

E-Mail : C.D.Turner@durham.ac.uk

2 February, 1993
Abstract

This report aims to outline the testing of object-oriented programs. At present there is little research being conducted, covering this area. A review of the current literature in the area is provided.

The testing process for object-oriented programs is compared and contrasted with the traditional approach of unit, and integration testing. The change of emphasis for testing from the routines themselves, to the testing of the interaction between routines via the data-members of a class is achieved by the application of a state-based approach which separately tests the definition and uses of the data-members. This new technique is described, and examples are provided for clarification.

This report assumes a basic knowledge of both object-orientation and testing; however, the majority of terms (those in bold) are usually explained in both footnotes (briefly) and in the glossary which appears towards the rear of this document.
Acknowledgements

Chris Turner is supported by a SERC CASE award in conjunction with British Telecom Laboratories (BTL), Martlesham Heath, Ipswich.

Many thanks to Mike Smith for his comments, and our discussions which helped guide my interest in this area of research.
## Illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1. An example of an instantiation tree (from the tool MKTC)</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2. An example of a class, showing potential data flow within the representation</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3. An Expanded view of figure 2</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4. A Finite State Automata for counting from one to four</td>
<td>19</td>
</tr>
<tr>
<td>Figure 5. A linked list class written in C++ by the author</td>
<td>25</td>
</tr>
<tr>
<td>Figure 6. The data scenarios for the features that act upon the nodes of the list</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7. The data scenarios for the features that act upon the links between the nodes</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8. An adaptation of the Von-Neumann model of processing</td>
<td>48</td>
</tr>
</tbody>
</table>
1. Introduction

The vast majority of testing research to date has been concerned with non object-oriented languages. Some research has addressed the problem of the testing of Abstract Data Types (ADTs), but the majority of this research is dependent upon the availability of formal specifications. This report will address this lack of research and review any currently available techniques.

A technique (to be) known as state-based testing will be described which addresses specific problems associated with the testing of object-oriented programs. Also guidelines for the adaptation of traditional functional and structural testing techniques will be provided.

The detail provided within this report is given in terms of class-based object-oriented languages. However, the techniques described are still applicable to non-class-based object-oriented languages (usually prototype-based languages).

The remainder of this document is as follows:

- The first section describes in detail the theory and the practice of the testing of object-oriented programs using information about the state of the object.
- The next section provides a survey of the current literature available that is concerned with the testing of object-oriented programs. This is compared and contrasted with the previous section.
- The next section provides a brief discussion of the applicability of more traditional testing techniques to the testing of object-oriented programs.
- The penultimate section briefly outlines a suite of tools that have been written to aid the user in the state-based testing of object-oriented programs.
• The final section summarises the information that is contained within this report.
• A glossary of terms used in this report is provided at the rear.
2. The Testing Process

2.1. Testing Background

This section describes the basic assumptions upon which the program testing within this report is based.

The very first, and possibly the most wide reaching assumption made during the testing of software, is that the compiler used to translate the source code into the executable program is correct, that is to say, that the compiler does not introduce any errors into the code itself. The testing of compilers is a separate area of research and therefore will not be discussed any further.

Validation consists of a number of distinct testing activities, such as unit, integration, system and acceptance testing. Of these, only unit and integration testing are within the scope of this report as the remaining activities have no special consequences for the testing of object-oriented programs.

Unit testing is the process whereby each separate unit of a program is tested in isolation. With imperative (procedural) languages, the chosen size of unit tends to be the function, or procedure; whereas, with object-oriented programs it is the feature (also known as a member function, method, or routine). However, it is difficult to isolate a feature from its surrounding class and then to write test drivers and stubs (which themselves require testing) to replace the code removed, when the majority of the code required is already provided in the form of other features of the class. From this, a conclusion can be drawn that the class is therefore the smallest unit of test.

Unit testing implies that test stub and test driver classes are to be written for the unit testing of the higher level classes. Following the same philosophy that was applied in the previous
paragraph for the unit testing of classes; the activity should be guided by the instantiation (or call) graph. The classes at the bottom of the graph which are called by other classes, but do not call any themselves should be tested first. The testing then continues up the graph using the previously validated classes in the testing of other classes. If the testing is conducted in this manner, the resources that would have been used for writing and testing the test stubs to replace the called classes, have been saved.

Integration testing is the combination of units that have been validated in isolation. The interface between the components is thus the focus of the testing process. If the values passed to other objects are monitored, then the interface between one object and another can be effectively validated.

If unit testing was performed as prescribed above, then the activities of unit testing and integration testing can be effectively merged. This merged process is of more benefit to object-oriented programming because of the difficulty in trying to write test stub classes which model the responses of other classes accurately without actually having to write the classes themselves.

There is of course an exception; when dealing with external stimuli, it is likely to be more cost effective to write the test stubs for unit testing. This choice has to be made by the tester, and must be based upon the resources required to write the stub as opposed to using the original class.

Figure 1 shows an example of an instantiation tree (taken from the tool MKTC which is discussed later). In the example, testing would start with the bottom layer, that is, STRINGOFTOKENS, STATEVALUES, SUBSTATETEREF; then continuing up the tree to LEX, SUBSTATES, ...etc.
This is an alternative to the more traditional view of the testing process. However, the technique which will be outlined later can be used with either views.

Functional testing is considered to be the testing of a piece of software with respect to its expected functionality (derived from its specification only). This has a tendency to be an overall system viewpoint. However, this document will use the term functional testing in the software component sense. It is the testing of a component against its specification, not the system specification. A test coverage measure (see later for a description) will be used to determine how much of the unit has, and has not been tested.

Structural testing is usually considered to be the testing of a program, or unit, against the specification, taking into account information derived from the program design and code. Any coverage that has not been achieved by the use of the functional test cases can be ‘made up’ by the use of structural test that are specifically designed to exercise those parts of the software which are unexercised. Both these views are consistent with those presented by Herrington et al. in [8].

Figure 1. An example of an instantiation tree (from the tool MKTC)
2.2. Emphasis of the Testing Process

At this point, the difference between classes and objects must be defined. For the purpose of this report, a class is taken to be the static, programmer-defined representation of an object. Therefore an object is a run-time instance of a class. Classes cannot be generally be tested directly; therefore it is actually the object created from the class that is tested. Nevertheless, the two are virtually synonymous when discussing the testing of object-oriented programs (written in class-based languages).

In procedural languages, the paths executed through procedures are determined by the parameters that are passed to routines, and the values of global variables. The parameters usually affect the control flow within the function and so produce the required outcome. However, in object-oriented languages the data representation of an object persists between calls of features of an object, therefore influencing the control-flow during calls to features.

Communication between features of the same class is facilitated by using data-members to store control-information\(^1\). The different values that these can store may influences the control-flow and thus the results of calls to features. These data-members can have their values set and/or used in any of the features. This is therefore the main emphasis for the state-based testing of object-oriented programs. This emphasis is in contrast with the data-flow testing of programs written in procedural languages which concentrates upon the definitions and uses of variables within the same function (during unit testing at least).

Figure 2 shows an example of the interaction (and therefore data flow) between the features of an object.

\(^1\) Control-information is data used to communicate events, rather than values, between features.
As can be seen in the diagram below (figure 3), the feature \texttt{first()} defines the value of the data member \texttt{bStart}, and the feature \texttt{next()} uses it. In data-flow testing, the path between the definition and the use would have to be exercised. However, a technique will be presented that separates the definitions and uses that exist in different features, for testing in isolation. The definition of \texttt{bStart} in the feature \texttt{first()} would be tested, and the result (the setting of \texttt{bStart} to true) would be verified. In addition, the use of \texttt{bStart} in the feature \texttt{next()} would also be tested (with both: \texttt{bStart} equals false, and \texttt{bStart} equals true). By performing the testing in this manner, the potentially random order that the features of a class can be called, is taken into account. The responses of all features to all states are tested, thus any incorrect responses would be detected. Also the possibly incorrect interaction between features would be noticeable.

When a class is defined, in the vast majority of cases, the only explicit (or implicit) order in which the features can be called insists that the constructor must be called before any other feature, and that the destructor must be the last feature called. Apart from that, there is likely
to be an infinite number of different possible combinations of feature calls. This possibly infinite order is tested by the aforementioned approach. It is achieved by exercising each feature with all possible starting states. The resultant states that can be generated by each feature are validated. It should not be possible to cause a feature to generate an invalid state. In addition, it is recommended that all features should be able to accept all legal states as input.

```plaintext
feature first()
begin
  bStart = true;
  ...
  ...
end

feature next()
begin
  if (bStart equals false)
    return;
  end if;
end
```

**Figure 3. An Expanded view of figure 2.**

State-based testing is testing of the interactions within a class by monitoring the changes that take place in the values of the data members. Therefore it tests the all of the definitions of data members within the features, along with all the uses. In addition, it must test for a lack of reaction by features to certain data-member values. Using this approach, 100% state coverage with a state-based test suite is attained by testing all the features of an object with all the possible states. Put in another way, 100% coverage is achieved by testing all possible state changes that can take place within an object, as well as all feature calls which do not affect the objects state.

---

2 A state is particular combination of values of all of the data-members of an object. It will be explained in greater detail later on.
In addition to state based tests, functional (specification) based tests must also be performed on
the class under test. Functional tests are important because they validate the external view of
the class, whereas the state based tests have a tendency to discard external results. For
example,

*When testing the 'get next token' feature of a lexical analyser class, unless
tests are extremely carefully designed (which is both tedious and time
consuming) so that the next token to be obtained would already be known in
fine detail, it is far easier to simply check for the effect on the representation
and ignore the actual token returned unless it is a special token e.g.
EndOfFile ...etc. Functional tests would be used to determine if the correct
token had been returned.*

This difference implies a contrast in emphasis between the two testing styles. State-based
testing places emphasis on the feature's correct interaction with the data representation of the
object; whereas, functional tests place emphasis on the correct interaction of the features with
the surrounding world (the rest of the program). Functional testing therefore concentrates on
the correct functioning of any calculations performed within the features as the first priority.

From this, two distinct purposes can be seen for the data representation: control information,
and data storage. Control information is used to flag various internal events to other features,
whereas data storage is used to store values required by the object. Data storage can be used
either to store information supplied by the caller, or to store the result of a call to another
feature, therefore allowing access to information that would otherwise be transient by nature.

The traditional testing process should consist of both functional testing and structural testing
[22]. It is deemed complete when the appropriate coverage level has been achieved. To reach
this level, first functional tests are performed, followed by the structural tests to increase the
coverage of specific constructs. This technique is easily applied to the testing of features,
however, this does not necessarily constitute a valid class.

C.D.Turner  -  14  -  02/02/93
Object-oriented programming is data-centred, that is, the functionality of the program is decomposed with respect to the types of data that will be dealt with, rather than simply decomposing the functionality into smaller tasks. To take account of the change in emphasis of the decomposition, state-based testing is required. Therefore the testing process for object-oriented programs should consist of three types of testing: state-based testing, and the two more traditional approaches.

State-based testing validates the model upon which the class is based, therefore if any major error in the functionality of the model is found, it is more likely to be detected during the state-based test than the other types. If an error is discovered, then the code has to be altered to correct the fault. The test cases that are dependent upon the code structure of the features may become obsolete, leading to the recommendation that the structural tests are not conducted until both the state-based testing and the functional testing has taken place.

If the features are being tested as part of a larger unit (that is, the class), rather than in isolation surrounded by test stubs and drivers then it is advisable to perform the state-based testing first. Once the interaction between the features has been tested, then the interaction can then be used as part of the functional tests, therefore exercising a greater part of the class before performing the structural tests. This causes a reduction in the test cases that are dependent upon the code structure of features.

2.3. Coverage Measures

Testing generally consists of both functional testing and structural based testing. The effectiveness of any test suite with respect to its testing method can be determined by the appropriate coverage measure.

---

3 Here, functionality refers to the correct interaction of all the separate section of the model, not the functionality of individual sections.
The aim of testing is to produce validated programs which conform to the specification as closely as possible. The only way to prove that a program conforms to the specification completely by testing is to perform exhaustive testing on it. Exhaustive functional testing may involve passing an infinite range of values as arguments and therefore it is infeasible in the vast majority of cases. Exhaustive path testing is also infeasible for anything but the simplest of programs because of the combinatorial explosion of paths that occurs as the program grows in size.

This introduces the requirement for a method for measuring the effectiveness of a test suite under the chosen testing method. The definition of 'effective' is very subjective, but is loosely related to the amount of program that is exercised. Therefore a measure of the program coverage is required. A number of different coverage measures exist, each one providing a differing level of validation.

Four examples are:

Statement coverage - Every statement in the program must be exercised at least once.

Branch (or condition) coverage - Every branch in the program must be exercised at least once. This implies that each conditional test within a program must produce both outcomes, that is, true and false.

Multiple condition coverage - Every clause in the conditional tests within the program must produce both outcomes, that is true and false.

LCSAJ\(^4\) coverage - Every LCSAJ (sub-path) through the code must be executed at least once. See [23] and [7] for more information.

---

\(^4\) LCSAJ is an acronym for Logical Code Sequence And Jump. In summary, it is a sub-path through the code that either starts at the beginning of the program, or the destination for a jump in the code; it continues until a statement which could cause a jump, or the end of the code is reached.
It is very unlikely that while using any of the above coverage measures, that 100% coverage of a program will be achieved. This is due to a number of factors:

- There may be branches within the program that may only be executed during exceptional circumstances.

- There may be parts of the program that cannot be executed at all. This may be due to a programmer's error, or simply due to defensive programming.\(^5\)

- If using a path coverage measure, then there may be paths through the program which are infeasible. An infeasible path is one which can never be executed; the most likely cause of these is the mutual exclusivity of two conditions. For example:

```
IF (x > 0) THEN
    .... // x is not altered during this branch
END IF
IF (x <= 0) THEN
    .... ENDIF
```

produces an infeasible path because the path that goes through both branches can never be executed.

Therefore a realistic coverage target must be set, based on the availability of resources for the testing, and an assessment of the code.

The different types of coverage measure facilitate differing levels of testing. If every statement in a program was exercised at least once during a test run, then there may still be faults in the program that lie undetected and will remain so until a specific path (or group of paths) through the code are executed. Branch coverage provides slightly more detailed testing, however, not as much as multiple condition coverage. Although exhaustive path testing is infeasible, testing

\(^5\) This is a technique whereby all conditions will be tested for, but some may never be achievable.
using **sub-paths** is not. Therefore the longer the sub-paths, the more 'effective' the testing method. The consequence of this is that LCSAJ testing (in theory at least) should provide a more effective testing method than multiple condition coverage.

A disadvantage of using a more effective testing method is the increasing difficulty in generating the test cases. However, this problem will not be discussed further as it is beyond the scope of this document.
3. State-Based Testing

3.1. Introduction

This section describes the process of state-based testing of classes and what it involves.

As its name suggests, the main emphasis of state-based testing is the values stored in the object's representation. The particular values are to be known simply as the object's state. This draws similarities from Finite State Automata and Infinite State Automata, which will simply be referred to as State Automata (or SA). SAs are objects whose next state is determined by the next feature that is called and the current state. However, some features do not affect the state, but simply allow a client to ask questions such as "what is your value?".

For example,

![Figure 4. A Finite State Automata for counting from one to four](image-url)
As can be seen in the above figure, the SA has four states corresponding to the numbers one through to four. There are four different features available for the client to call (including a constructor) and their effects are shown above. The feature `GetValue()` does not affect the current state, it simply informs the client (caller) what the current value is. The `Reset()` feature allows the state to be returned to the start (one) at any time.

The counting SA would be relatively trivial to test, especially as all features are defined over the whole range of states of the objects with no exceptions. Each feature would be called with the object in each of the four states. The result of the call would be verified against the diagram above.

Any feature has four types of possible responses to a particular state and feature argument combination:

1. It can change the object's state to the appropriate new state.
2. It can leave the object's state as it is.
3. It can change the object's state to an undefined state (error).
4. It can change the object's state to an inappropriate state (error).

Of these, iii) and iv) are definitely errors, and ii) can be an error if the object's response was supposed to be i). Therefore the aim of state-based testing is to detect all occurrences of the erroneous ii), iii), and iv).

Chow discusses testing using SA's in [2]. However, he discusses the testing of control-structures that can occur within a design, rather than an implementation. His approach of predicting the correct response of a routine according to the SA upon which it is based, is similar to the one described in this document.
3.2. Notation

To introduce a minimal degree of formalism, each feature is considered a mapping from its input states, to its output states, affected only by the current state of the object, and any parameters passed\(^7\). The notation to be used for expressing this is as follows:

\[
\begin{align*}
\sigma &= \text{The set of valid states that the object can have.} \\
\varepsilon I_i &= \text{The set of states that the } i\text{th feature is expected to handle (by both action, and inaction) as input}\(^8\). \\
\varepsilon O_i &= \text{The set of states that the } i\text{th feature is expected to be able to produce as output}\(^9\). \\
\alpha I_i &= \text{The set of states that the } i\text{th feature actually accepts as input.} \\
\alpha O_i &= \text{The set of states that the } i\text{th feature is actually able to produce as output.}
\end{align*}
\]

If a class is valid (as far as state-based testing is concerned), then \(\alpha I_i = \varepsilon I_i\) and \(\alpha O_i = \varepsilon O_i\), for all the features of the SA. In essence this simply means that all the states each feature accepts as input are as expected, and the same for the output states.

However, the previous example is only a simple one requiring only a single data member needed to store the state of the object. When applying this type of testing to classes, a number of factors influence the effectiveness of the process. Firstly, the number of data members\(^10\) that the class possesses has a profound effect on the complexity of the testing. The more data members, the more time it takes to design and implement the tests. Secondly, different types of classes exist, each providing differing levels of emphasis on the data representation. The more features that inter-act with the representation, and each other via the representation, then the

---

\(^7\) If global variables are accessible, they are treated as if they were passed as a parameter.

\(^8\) Input is used to mean the current state of the object before the call to the feature.

\(^9\) Output is used to mean the state of the object after the call to the feature.

\(^10\) Data members is the term used in the C++ community to describe the variables that make up the data representation of an object.
more effective the state-based testing process will become. Thirdly, the greater the code complexity\(^{11}\) within features, the more difficult those features are to test. This can be circumvented to a certain degree by inserting at the end of each sub-task within the feature an assertion to check the state of the object (this will be discussed in more detail later).

### 3.3. Substates

The current state of an object is the combined values from all of its data members at the current point in time. It is appropriate to therefore define a substate to be the value of a particular data member at a specific point in time; likewise, a partial state is the combined values of a subset of all of the data members of the class at a particular point in time. This means that the state of an object can now be redefined to be the combination of all of the substates of the object at a point in time.

Rather than associate every single possible value of a data member with a substate, it is more appropriate to introduce two types of substates values:

- **specific substate values** - these are substate values that are tested for directly within the code, or are described in the design as being of special significance.
- **general substate values** - these are a group of substate values that are all considered in the same manner; therefore there is no need to distinguish between them for the purposes of state-based testing.

This means that a data member's substates values are now a collection of specific substate values and general substate values. For example,

- given an integer data member `iValue`, and a design that states:
  - -1 has special meaning

---

\(^{11}\) The term complexity is used to imply that multiple tasks are performed within the feature. An implication of greater complexity is that there is more likely to be a greater number of state transitions.
that the value is always greater or equal to -1
and all numbers greater than minus one are treated equally,
then iVar can have the following substate values:

• StartValue (iVar = -1) - a specific substate value
• RemainingValues (iVar > -1) - a general substate value

To determine the substate values for a class, perform the following for each data member:

1) analyse which particular values are significant (by analysis of the code, or the design),
and which are not.
2) Allocate one substate value for each significant value.
3) Allocate one substate value for each group of related values.

If the data member being considered is a pointer (an address of an object, rather than an actual
object), then it is most likely that the pointer will only have two substate values, one specific
and one general. The specific value is likely to be the value NULL\textsuperscript{12}, and the general value will
cover all other possibilities, that is, not equal to NULL. However, it is possible that the
substate can have more substate values, such as the address of a specific object.

An example of substates generated for a class is given in the section headed "Data Scenarios”.

3.4. Performing the testing

The next part of the testing preparation is to determine the $\varepsilon_i$ and $\varepsilon'_i$ for all the features of the
class. This must be done from the design of the class, not from the code, as there may be errors
in the code.

In order to facilitate the testing of substate-values, extra features must be added to the class
under test. This will be described below.

\textsuperscript{12} Null is generally used as the value of an invalid pointer in C/C++.
3.4.1. Additions to the class

A major part of state-based testing is the determining of the object's current state. To enable this, a new version of the class under test must be produced. At least one new feature per substate is required. These will enable the tester to inspect the value of chosen substates. However, if there is a state change between two values which were both considered to be part of the same general group, then the change is undetectable. This problem is easily rectified by the addition of an extra set of data members, whose purpose is to mirror the value of the original data members that the user is interested in. For each data member being mirrored, an extra feature is required to test the difference between the original and the mirroring data member, and to update the value of the mirror data members. This is explained in greater detail in a separate report describing the MKTC suite of tools [19].

As a simple recommendation, it is advisable to insert statements at the beginning and end of each feature to report to the screen (or to a file) the value of the parameters passed and the values returned by the features. It is useful if this is also done for the substate testing features, providing a useful aid in debugging any errors that might occur, by allowing the execution of the test case to be traced.

It is usual that some features of a class must perform more than one task to achieve their desired functionality. The greater the number of tasks, the more difficult they will be to test and debug if errors are present. As a recommendation to aid the tester, it is advisable to insert assertions\textsuperscript{13} into the code between each task, checking the individual task performed as well as the feature as a whole.

\textsuperscript{13} Assertions are statements about the current value of variables, or substates.
3.4.2. Data Scenarios

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

For Example,

class\textsuperscript{14} list
{
  public:
    // ...

protected:
  // this structure creates the links of the list
  struct list_element
  {
    list_element * pNext;
    TYPE tItem;
  };

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

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

\textit{Figure 5. A linked list class written in C++ by the author.}

\textsuperscript{14} This class was actually written as a template (a generic class), but for simplification reasons, it has been reduced to a class which uses the class TYPE for its elements.
With the following substates:

S1) pTop
   i) pTop equals NULL
   ii) pTop does not equal NULL
S2) *pCur
   i) *pCur equals NULL
   ii) *pCur does not equal NULL
S3) pCur
   i) pCur equals the address of pTop
   ii) pCur does not equal the address of pTop

The significance of these states are:

- if S1(i) then the list is empty
- if S2(i) and S1(ii) then the current node pointer is past the end of the list
- if S3(i) then the current node pointer points to the top of the list

In conjunction with the model upon which the class is based (not shown), two different sets of scenarios are produced: scenarios used by those features of the class that act upon the elements themselves (see figure 6), and the scenarios used by those features which act upon the links between the nodes (see figure 7). Generally, the type of features that fall into the first category are those that alter the contents of nodes, but do not actually change the structure. Whereas the type of features that fall into the second category are generally those features that add, or delete nodes from the list.

---

15 A star in front of a variable name in C/C++ means the object pointed to by the variable (a pointer). In this case, it means the current element in the list.
when put into words, they are:

a) the only node of the list,

b) the first element of a multiple node list,

c) the last element of a multiple node list.

*Figure 6. The data scenarios for the features that act upon the nodes of the list*

When put in to words, they are:

a) an empty list,

b) before the only element,

c) after the only element,

d) in between two nodes in a multiple node list,

e) at the end of a multiple node list.

*Figure 7. The data scenarios for the features that act upon the links between the nodes.*
These scenarios are used as additional test cases to the appropriate groups of features. However, unless a change in the composition of the data structure is detectable, the results of using the above scenarios will be difficult to verify. Therefore, additional substates are required for detecting the state of various parts of a dynamic data structure. For the above example, the following additional substates were used:

S4) (*pCur)->pNext<sup>16</sup>
   i) invalid
   ii) (*pCur)->pNext does not equals NULL
   iii) (*pCur)->pNext equals NULL

S5) (*pCur)->tItem
   i) (*pCur)->tItem is invalid
   ii) (*pCur)->tItem is valid

The substate values S4(ii) and S4(iii) are relatively self-explanatory<sup>17</sup>; S4(i) has to be included, because there is a possibility that *pCur could be NULL rendering the expression (*pCur)->pNext invalid. This also applies to S5, which has an added anomaly, it seems to indirectly reflect the value of *pCur (S2) only. This is because a list is used to store any range of values; it is pointless to detect every single value, therefore what is required is the ability to detect a change in value. This is provided by some of the extra features that have been added to the class under test; but for identification purposes, allocating a substate to the expression eases some problems. It also facilitates the correct expression of test cases to the tool MKTC<sup>18</sup>.

---

<sup>16</sup> The arrow ‘->’ is used in C/C++ to dereference a pointer to give an object, and then to access an attribute of the object. In this case, it refers to the pNext data member which is part of a list_element object.

<sup>17</sup> They reflect whether the pNext data member of the current node points to another link, or to NULL (the end of the list).

<sup>18</sup> MKTC will be described briefly later on in this report.
When choosing substates for use with the data scenarios, it is not necessary that they should be able to differentiate between the scenarios; the initial set-up of the test case is driven by the tester, therefore allowing the starting state to be whatever is required. It is for this lack of differentiability between scenarios that the tool MKTC has a separate facility allowing the starting state for test cases to be expressed as a scenario, rather than a state description\textsuperscript{19}.

### 3.4.3. Generation of Test Cases

In this report so far, both substates and data scenarios have been introduced and can now be combined to provide a more general guide to state-based testing. The guide is in the form of a list of tasks to be performed.

1. Allocate one substate per data member of the class under test. This is done as described earlier.

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

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

4. Determine the specific-values and the general values for all these substates. Include the invalid values that are required when de-referencing a pointer as mentioned in the previous section.

5. Add the features to test the for the substate values to the class.

6. Determine which substates a test for a change of value is required. Add features to detect the changes to the class under test.

---

\textsuperscript{19} A state description is a list of substates descriptions which when combined form the state of the object. A substate description is the declaration of a particular substate's value.
From the design of the class, determine which states are the $I_i$, and the $O_i$ for each feature. In the majority of cases, both should be the same as $\sigma$ because classes should be written with no implied order for the calling of features [13]. These sets of states form the basis of the cases.

Analyse from the design the call graph for inter-features within the same class, that is, the class under test.

Start with the features at the bottom of the graph, especially those features which are not called by any other feature within the class under test.

For each state $s_i$ such that $s_i \in I_i$, calculate from the design which state $s_o$ such that $s_o \in O_i$ for feature i (That is, for each state that the feature is expected to handle as input, calculate the state that the feature should leave the object in after the call). This is done for all significant values that can be passed as parameters to the feature. These significant values are the stimuli of the features.

Generate the code to create the test case scenario, that is, the starting state of the object. Follow this with test to validate the starting state of the object. After this, add the code for the test, including the call to the feature under test. Append to all of this, a test for the final state of the object, and any code required to tidy up after the test.

Go back to item 10 until there are no more features left to test.

3.5. General Assessment of the Technique

State-based testing, like the majority of other techniques, has particular types of classes on which it is more effective than on others. The types of classes that have been tested by this technique so far have by no means been restrictive. They include file handlers, parsers, lexical analysers, and abstract data types such as lists. The degree of effectiveness of the technique is dependent upon the degree of control-information that exists as part of the data representation.
of the class. The more the features of a class interact via the representation, the more effective state-based testing is likely to be.

However, if the class is designed simply as a repository of information, or as a non dynamic data storage structure, then this technique will have limited effect. Its ineffectiveness is related to the number of test cases that can be produced. From the experience of the author, the number of test cases that can be generated is related to the amount of control-information present within the representation of the class and the number of different substate-values for each data-member.

The technique is not a substitute for functional testing, nor structural testing. However, it should be considered as a complimentary technique for testing that part of programming that is particularly prevalent, and easily accessible in object-oriented programming, but which is not so readily visible in programs written in more procedural languages.

If a class is written with no order imposed upon the execution of features, the less restrictive it is upon any programmer who may to make use of it. This in turn increases one of the aims of object-oriented programming - that of facilitating the increased use of reusable components, therefore reducing the cost of software production [12].

There is one point which has been carefully avoided until now - the emphasis of the test cases. The dilemma is: should the test cases concentrate on individual substates, or should they attempt to concentrate on combinations of substates?

There does not seem to be a simple answer to this problem. If emphasis is placed on individual substates, the number of test cases increases significantly, which although significantly faster and easier to generate than the other type (when using automated methods such as MKTC), but do have the disadvantage of using more testing resources such as machine time, and disk space. However, it is possible to find a compromise between the two, and this must be decided upon by the tester.
4. Other Testing Techniques for Object-Oriented Programs

This section will review the other work that has been conducted in the testing of object-oriented programs. As there only a few papers, each will be discussed individually and contrasted with the rest. They are discussed in chronological order.

The first paper is "Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience" by Walter G. Olthoff [15]. It described an approach to the validation of object-oriented programs written in a modified version of Pascal that includes constructs for defining classes, and for the inclusion of information connected with the ADT theory.

The author gives a brief introduction to the various different methods of formally specifying software components. He then concentrates upon the algebraic specification method, which is the one that has been used in conjunction with the modified Pascal. Details of the modified Pascal language are then given including the declaration of classes, enrichment objects (extensions to other objects), and the incorporation of information from the formal specification. The main problem with the use of a 'home-grown' version of any language, is that it is very unlikely that there is likely to be any third party tool support. This restricts the user to the use of the limited tools that are supplied as part of the supporting environment.

The use of formal specifications by the system poses problems for the tester. They require an experienced mathematician to be able to manipulate them with any degree of competence and success. At present very few tools and techniques are available to aid the programmer in the production of the specifications.

Towards the end of the paper, the author discusses whether it is appropriate to model ADTs in imperative object-oriented programming languages. He concludes that it is "not useful to remodel the theoretical notion of an ADT in an imperative [traditional object-oriented]
language." However, he makes a number of other claims about the applicability of ADT to imperative languages, including that the specification and concrete implementation of an object should be done using separate language constructs.

Steven P. Fiedler in "Object-Oriented Unit Testing" [4], describes the application of a more traditional approach. He describes experience gained on a software project at Hewlett Packard. The project applied traditional testing techniques to the testing of software written in C++. When describing the testing process, the author states:

"Since the paradigm of object-oriented programming emphasises the external behaviour of data abstractions rather than the internals, one would only expect to employ only black box, functional testing techniques. However, a more robust testing structure employing complete path testing is actually needed."

The McCabe cyclomatic testing method was used to ensure independent path coverage of methods. Test cases were derived from the range of values expected by parameters to methods, boundary analysis of parameter values, and erroneous values to place the unit under stress. Other test cases were derived from the unit's specification.

Classes are validated in the order determined from a call graph of the project. The classes at the bottom of the graph are validated first, removing the requirement for test driver classes which emulate unvalidated parameter classes. This is a similar conclusion to the one reached earlier in this report.

Results from the test runs are verified by simple file comparison with the expected results. This technique is cumbersome because of the time required to generate the expected output exactly. However, it is an easier technique to use, rather than to develop self-verifying test cases. It is suggested that rather than using a team of testing personnel separate from the team of developers, it is more efficient to allow the developers to test the code because of their detailed knowledge of the internal workings of the unit. This decision is likely to be based on the availability of resources for the testing. However, it has been suggested that although people
who do not know the code will find it more difficult to generate the test cases, they will have fewer preconceptions about the performance and functionality of the unit under test, and therefore are more likely to detect errors. In [20], Wallace and Fujii imply the separation of the validation and verification group (who perform the testing) from the development group.

When discussing the testing of classes defined using inheritance, it is stated that the methods of the class that are provided by a parent class only require minimal testing. This is a rather simplification of the problem concerning the reduction of testing required by derived classes. This problem has been addressed in greater depth by the work conducted by M.J. Harrold et al.

It is concluded that a significant investment of time is required for the testing described, although accurate estimates of time required for the testing were not available, because testing and development occurred concurrently. Testing methods designed for use with procedural style languages can be adapted for use with the object-oriented paradigm, although they have to be applied to classes as a whole unit, rather than to individual methods. However, it is more practical not to test features in isolation, but to simply concentrate the test cases on exercising each feature in turn to its maximum coverage. This will be addressed further, later on.

Dewayne E. Perry and Gail E. Kaiser in "Adequate testing and object-oriented programming" [16] discuss the theoretical view of the testing of object-oriented programs. The authors describe the applicability of Elaine Weyuker's test adequacy criteria (described in [21]) to the object-oriented paradigm. Originally designed for the procedural style of programming, the criteria are used to determine if a program is adequately tested using the chosen technique. The paper is based on the thesis contrary to current belief, that well-tested classes must be retested in the new contexts created by the derivation of new classes. This point is contrary to the algorithm produced by M.J. Harrold (see later).

In the paper each of the original criteria are listed, along with a description of their applicability to object-oriented programming. Any effects on the testing of programs are also noted, especially when seemingly counter-intuitive. Comprehensive examples are provided in an attempt to clarify the effect of each criterion. Seven of the eleven criteria are "intuitively
obvious” and/or have no special effect on object-oriented programs. Only the criteria relevant to this report will be described.

The criteria affecting object-oriented programming are:

The antiextensionality criterion - (summarised) if two programs compute the same function, a test set adequate for one program is not necessarily adequate for the other. This implies that functional testing is not the only testing method performed. Structural test cases are dependent upon the implementation of the functions, and therefore may not be adequate for the other function.

The General Multiple Change criterion - (summarised) if two programs are syntactically similar, a test set adequate for one program isn't necessarily adequate for the other. This criteria implies that structural testing is not the only testing method performed. Just because two functions have similar structure does not mean that they compute the same function.

The Antidecomposition criterion - (summarised) A program component is likely to require retesting, if it is taken out of the context of the surrounding program.

The Anticomposition criterion - (summarised) Testing individual program components in isolation may not adequately test them because interactions may occur after composition that could not occur in isolation. This implies that a path coverage structural testing technique must also be used. When two components are combined, the number of paths through the two components does not simply double, it increases combinatorially. It is intuitive (but made explicit by the criterion) that when a base class is modified, all classes derived from it must be tested along with the base class.

An effect of the antidecomposition criterion which is counter intuitive, with major ramifications for the reuse of software, is the requirement for all inherited features to be retested. This is because the context of the features might have changed. However, if a feature is adequately tested as part of the base class, the only problem that can occur is if a derived class has access to the data-representation of its parent class. This should be tested when the derived class is
tested, therefore this assertion is less valid. It is also dealt with in the paper by M.J. Harrold (see later).

The overriding of methods (that is, where a child class replaces a method provided by a parent class) is covered by the antiextensionality criterion, which intuitively states that the test set for the old version of the method is unlikely to be adequate for the new version.

If the implementation of a class is changed, but its functionality is still the same, it would be expected that only the modified class requires retesting. The anticomposition criterion is contrary to this, suggesting that all classes that use the modified one, must also be retested. However, dependencies between classes are usually explicit and therefore client classes should not require retesting; otherwise there is no incentive for producing modularised programs. If classes are connected via global objects, then there is a requirement for retesting.

The authors conclude the paper by stating:

"Inheritance is one of the primary strengths of object-oriented programming. However, it is precisely because of inheritance that we find problems arising with respect to testing"

They also state that encapsulation, instead of reducing testing problems, compounds them. However, if interfaces are defined accurately between program modules (classes) encapsulation allows a program to be tested in smaller units. In general, the smaller the unit, the easier it is to test and debug. It is felt that this does not compound testing problems, although the use of global variables does.

The units must also be tested together to ensure that the units connect correctly, but this can hardly be construed as compounding testing problems.

Their work is based upon earlier work conducted by Elaine Weyuker on a set of principles for assessing test effectiveness criteria\textsuperscript{20}. The original principles were to be used for the

\textsuperscript{20} The criterion are used to detect when a program has been adequately tested, hence testing need not continue any further.
comparison and evaluation of new criterion, whereas the authors have considered them from a different view point.

A description of some of the problems that might be encountered when considering the testing of object-oriented programs is given by M. D. Smith and D. J. Robson in "Object-Oriented Programming - the Problems of Validation" [17]. They discuss validation (in both the development and the maintenance phases of the software life cycle) with respect to object-oriented programming. The authors comment that industry sees object-orientation as the solution to all of its problems (the great white hope), without noticing the new problems introduced.

Because a class is a mapping of the real world into code, it is suggested that more test cases may be intuitively derived, due to an ease of visualisation of the model on which the class is based by the programmer and tester. However, because functional test cases are derived from the specification, this statement has no true foundation. It is suggested that especially during the maintenance phase, inheritance could possibly increase the testing required. The authors state that strict inheritance\(^{21}\) is the easiest to test because the test cases for the parent class are still valid for the derived class. Extra test cases are required for the added or redefined methods and to check that the parent's representation has not been invalidated. This conclusion is similar to part of the algorithm developed by M.J. Harrold. The author discusses various problems with multiple and repeated inheritance, genericity, and polymorphism.

The authors state that

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

They then discuss the applicability of data-flow and control-flow testing methods. As a conclusion, the author states that there is no way of specifically testing the inter-class

\(^{21}\) Strict inheritance is where all the features of a parent are inherited by the derived class.
relationships, or any method for testing the validity\textsuperscript{22} of the inheritance itself. The inter-class relationships between classes are tested by the integration testing of the classes. However, the validity of the use of inheritance is a subjective decision and therefore, can be tested, but only manually.

They also state that there are no accepted methods for testing the main constructs offered by object oriented programming, including the constructs provided for the reusability of software. These views are contrary to the opinion of Fiedler [4] who describes experience of testing object-oriented programming by applying a technique originally designed for the procedural style of programming. This point will be addressed further, later on.

Another paper describing the use of formal specifications in the validation of object-oriented programs is "Tools for Object-Oriented Programs" by Phyllis G. Frankl and R. K. Doong [5]. It describes A Set of Tools for Object-Oriented Testing (ASTOOT), written in Eiffel. Their approach is based on the algebraic (axiomatic) specification of ADTs. Test cases are generated, consisting of pairs of sequences of methods (referred to as operations or methods in the paper) along with a tag. The tag reflects whether the two sequences of methods should leave an object in the same abstract state. The method depends heavily upon the completeness of the ADT's specification (see [10] for more details).

A class is said to be a correct implementation of an ADT if and only if, for every pair of sequences s\textsubscript{1} and s\textsubscript{2}, applied to a pair of objects, o\textsubscript{1} and o\textsubscript{2}, the objects are put into observationally equivalent states. A user supplied routine (EQN) is used to check the equivalence of o\textsubscript{1} and o\textsubscript{2}. The result from EQN will be compared with the tag associated with the two sequences. EQN can be either semi-automatically derived from the specification, or hand-crafted.

---

\textsuperscript{22} The validity of inheritance refers to the validity of the use of inheritance to define the child class in terms of the parent class, that is to say, is it a true "is-a" relationship, or is it simply expressing implementational detail.
ASTOOT consists of three components; the driver generator, the compiler and the simplifier. The compiler and the simplifier combine to form an interactive test generation tool, allowing the user to generate a sequence of operations from an algebraic specification. The simplifier uses rewrite rules provided by the compiler to produce two equivalent sequences of operations which form a test case. The driver generator allows the execution of automatically or manually generated test cases.

The Specification language used (LOBAS - Language for Object Based Algebraic specifications) allows a restricted form of inheritance to be expressed in a specification. If an ADT \( T_d \) is expressed as a descendant of an ADT \( T \), then it inherits all of \( T \)'s methods. The methods of \( T_d \) are constrained by the axioms of \( T \) and can be constrained further by the addition of more axioms. It is stated that not all test cases for \( T \) must be rerun for \( T_d \), only the test cases for redefined methods need be rerun. This philosophy is founded in specification testing; it is only testing the functionality of the class (as defined by the specification). Because classes are not automatically derived from the specification, A class can only be a true implementation of an ADT if exhaustive testing is used. Specification based testing only tests the external view of the class without any regard for its structure or implementation. The authors state that test cases which consist of two sequences with unequivalent results, in addition to test cases with equivalent results enhances the testing of programs.

The work discussed earlier by Perry and Kaiser shows considerable shortcomings with the work by Frankl and Doong. Because their method is purely specification based, it must be used in conjunction with structural techniques to satisfy all the criteria discussed by Perry and Kaiser.

The approach makes no direct observation of the exact state that the objects have reached after execution of the test sequences. Although this allows the same set of test cases to be used for different implementations of the same ADT, Fiedler [4] states that although the object-oriented paradigm suggests functional testing is required, a more robust testing method including complete path testing is actually required.
When comparing their work to others, the ability of their test execution tool to accept hand-crafted scripts for situations when a formal specification wasn't available, was seen as an advantage. As the whole approach is based around the functionality defined by a formal specification, without the axioms being used to derive test cases the approach has no advantage over more traditional functional/exhaustive testing methods. The test case generation is user-driven at present, requiring a large amount of time to generate the 2771 test cases the authors used to test a heap implementation.

Further enhancements are planned for ASTOOT, including the use of testing criteria to drive the automatic derivation of test cases. It is hoped a symbolic evaluator can be used to discover the values within the axioms of the specification, providing the opportunity for the automatic derivation of more test cases.

T.J. Cheatham and L. Mellinger in "Testing Object-Oriented Software Systems" [1] claim to address the testing of object-oriented programs at the unit level with respect to message passing and inheritance. However, the paper simply mentions some of the problems, and then without justification, states that with a Smalltalk library of tested classes, 40-70% of unit testing is 'free'. No references or details of a study is provided to substantiate this. A simple algorithm for determining which features to restest when testing a derived class is provided. It is in agreement with the algorithm produced by Harrold (see later), but is a simplified and less rigorous version.

The author mentions that because of the tight coupling between the model on which the program is based and the real world problem, system level test cases become easier to design. However, the system level is the view of the program as a whole, not as individual components, thus the design and implementation techniques are of little consequence to the design of test cases at this level.

The author argues for bottom-up testing in much the same fashion as was done earlier in this report. He discusses both unit and integration testing without actually defining how it is applied to object-oriented programs.
The article is concluded by saying that more research is needed into the testing of object-oriented systems.

As a follow-up to their earlier article [5], Roong-Ko Doong and Phyllis Frankl in "Case Studies on Testing of Object-Oriented Programs" [3] provide experimental details of their method and tools. The first section of the paper just gives a simple introduction to object-oriented programming, and defines the terms that are used throughout the remainder of the report.

They state that little work has been done in the area (as do all the others), and then go on to discuss the testing of features in isolation. They conclude that the testing of features as a mapping from an input space to an output space shifts the focus of testing away from the essence of data abstraction (the interactions between features). Their ASTOOT tool is dependent upon algebraic specifications which specify a class by the interactions between its features, therefore requiring this point of view. The testing of object-oriented programs should embrace all methods of testing, from the state-based approach, through functional testing techniques (such as this one), to structural testing techniques (such as the described by Fiedler in [4]).

The problem which the method they propose (described earlier), is more of a problem with the specification technique chosen, rather than the method itself. Some classes cannot be specified by simply describing the interactions between the features, and therefore would not be suitable for testing with this approach.

They contrast their method with another one based on formal specification of abstract data types, called DAIST. Because their test cases can result in both equivalent, and unequivalent states, they conclude that their method is likely to discover a greater number of errors that DAIST.

The next section describes an experiment they conducted with two example ADTs which both had a 'purposeful' error seeded into them. A number of questions were asked during the
experiment such as "How does the length of the sequences used a test cases, affect the detection of errors?" Their results causes two conclusion to be drawn:

- Larger parameter ranges with longer sequences of operations seemed to be more effective.
- Smaller parameter ranges with longer sequences of operations seemed to be less effective.

The authors then go on to discuss the testing of inheritance with their method. They outline a restricted form of inheritance whereby a derived class still satisfies all the axioms of its parent class, even if a feature is redefined. However, they introduce an exception - if a derived class is simply expressing code reuse, rather than conceptual reuse, then the derived class does not have to conform to the parent class' axioms.

The paper is concluded by discussing that additions that it is hoped will be made to the ASTOOT tool suite in the near future.

The first paper to consider the testing of class hierarchies is "Incremental Testing of Object-Oriented Class Structures" by Mary Jean Harrold et al. [6]. The paper describes an incremental algorithm for the testing of classes within an inheritance hierarchy.

They state:

"[inheritance] permits a subclass to inherit attributes\textsuperscript{23} from its parent classes and either extend, restrict or redefine them. Subclasses may inherit attributes from a parent class, cancel attributes not possessed by the parents, and/or redefine some of the parent's attributes."

This statement forms the basis of their algorithm which aims to reduce the number of test cases which need to be rerun when one class is derived from another. This is contrasted against the

\textsuperscript{23} Attribute is a generic term used to refer to both the features, and the data members of a class.
other method that they suggest, that of flattening the class structure - each class is tested as if it provided all of the features.

The suggested order of testing for an inheritance hierarchy of classes is (intuitively) from the top downwards, that is, starting with the base classes, and continuing down the hierarchy. Testing must proceed by testing each feature in isolation, and then testing the interactions between the features. State-based testing allows the interactions between the features to be testing in isolation, and therefore could easily be incorporated into this algorithm at a later date.

The algorithm they suggest makes use of the test histories from parent classes. A Test history associates a test case with the features it test. A test history for a derived class is created by incrementally updating the test histories from the parent classes with information about the derived classes differences from the parent(s). From the new test history, reusable test cases from the parent class can be identified along with any attributes for which new test cases must be generated.

The authors describe the language model upon which the algorithm is based. It includes 'parameterised inheritance mapping" taking account of the private and public declarations that are available in C++ for base classes. It also supports three levels of attribute visibility: hidden, accessible to derived classes, and accessible to all classes. They then introduce six types of features covering all ways of defining a feature in a derived class from inheriting it, to defining it afresh.

They introduce the work done by Perry and Kaiser [16] (see above). This provides their justification for the testing of features both in isolation, and in interaction. For each of the four test adequacy criteria of special significance to object-oriented programming, they briefly describe how they are satisfied by their algorithm.

Inheritance is remodelled in terms of a class and a modifier, thus producing a derived class. The modifier thus forms part of the incremental update that is applied to the test histories of the
base classes. They state "The modifier approach permits a decomposition of the inheritance structure into overlapping sets of class inheritance relations." Examples are provided to illustrate this point.

Very little information is provided for the actual generation of the test cases, nevertheless, the paper states that both functional and structural test suites should be produced for both the isolation testing of features, and the integration testing of features. Examples are provided showing how the test suites are reused for the derived class.

The only major omission from the paper is a demonstration of the algorithm demonstrating that the derived classes are still 100% valid even though some test cases have not been rerun because some features have been inherited without change from the parent class(es).

The authors of the paper are using a dataflow analyser added on to the Free Software Foundation Inc.’s G++ compiler. It is then use for experimentation with the technique.

Another paper describing the use of formal specification for the verification of object-oriented programs is "Modular Specification and Verification of Object-Oriented Programs" by Gary T. Leavens [11]. It outlines a method of reasoning about object-oriented programs using relationships between the ADTs used. ADTs are referred to as 'types'.

If given two types, A and B; if B is the subtype of A, then A is the supertype of B. Type B must provide a similar behaviour to that of A. Because of this restriction, the supertype can be used in the reasoning about any of its subtypes. This (theoretically) reduces the level of verification that has to be performed. The technique is known as supertype abstraction.

The author stresses that subtypes are not sub-classes. Subtypes are a specification issue whose constraints are based upon behaviour, whereas a sub-class is generally restricted by the inheritance of its parents implementation details. This causes problems within the majority of object-oriented languages that can be used for the implementation. There will not be any direct correlation between the types and subtypes used during the specification and the classes used
during the implementation. The author addresses this problem by suggesting that future language designers should consider separate sub-classing from subtyping.

For every type specified, a separate 'trait' must also be specified which describes abstract values for the objects of that type; for example, an empty set.

The verification aspect of the paper is concerned with the design and specification of the program, rather than the validity of the implementation. To facilitate the verification, the relationship between a type and its subtype is subjected to a number of constraints. If a subtype has a number of supertypes, then it must satisfy the constraints of all of its supertypes.

The author addresses the verification of an implementation by discussing the need for an abstract relation $A$ that links objects created by a module to their abstract values. A module is verified by demonstrating that all operations satisfy their specified preconditions and postconditions. However, this particular subject is not discussed in much detail.

Syntactic constraints applied to the specifications ensures modular specification. However it does not ensure modular verification. The author the goes on to describe the major problem with the modular verification of types and their subtypes. A subtype's preconditions for a particular message are likely to be textually different, or if they are the same then they are likely to have slightly different meanings. His solution to this is to specify the subtype so that there is a relationship (known as the simulation relationship) between the type of the actual arguments and the type of the arguments that were used during verification. For example, Integer Interval$(i,j)$ simulates Integer Set$(i \text{ to } j)$.

The simulation relationship must preserve the validity of assertions (preconditions and postconditions). They must be constructed so that every object of a subtype simulates an object of each of its supertypes. If an object has many supertypes, this will be a considerable task.

The whole technique revolves around the simulation relationships that are defined between a subtype and its supertypes. These relationships however, are not formally defined, therefore they can only be expanded exhaustively, that is, object by object. To describe the relationship
between two types exhaustively is very rarely possible because of the infinite number of abstract values possible, thus reducing the applicability of this technique.

The author concludes the paper by stating the requirement for more formalism in the specification of the relationships between a supertype and its subtypes. Because the subtypes that are likely to exist in programs go far beyond the bounds of simulation then the described technique can only be used informally to check whether the desired sub-type relationships have been achieved within the design.

The final paper to be discussed, is "A Framework for Testing Object-Oriented Programs" by M.D. Smith and D.J. Robson [18]. It describes both a method and a framework designed for the testing of object-oriented programs.

The authors discuss the terms that are normally applied to the testing of software written using traditional approaches. They then redefine the terms for the testing of object-oriented programs. Testing is defined as:

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

The authors then go on to describe the Von-Neumann model of processing as a passive store with an active processor. The authors then claim that object-orientation is quite the opposite, with a passive processor and active data. However, there is an alternative view to this which would allow the use of traditional testing techniques (instead of excluding them). If the authors were instead to consider the feature as a possible unit in object-oriented programming, the Von-Neumann model would simply be adapted to:

input + current state -> process -> output + new state.

The authors different view point is also used when discussing the execution of test cases. They claim that software is no longer executed sequentially, with the features of a class being
combined in any order. However, even in traditional programming languages, routines could be combined in any order; object-orientation simply provides a different emphasis during the program design and construction.

The authors discuss a number of points which must be minimised during the testing process. These include the effort required, in both the production of test cases and the execution.

Because of the above point of view, the authors feel that functional testing is not applicable. Also they feel that structural testing is not directly applicable as it is difficult to analyse the control or data flow through a class. However, functional testing can be done using the method described by Frankl and Doong [5]. Structural testing is applicable without major modification at the feature level, an example of which is described by Fiedler [4], thus invalidating this claim.

The remainder of the paper describes the structure and operation of the framework. This includes the test strategies which are used to guide the actual testing of a class. They are object-oriented in the sense that they can be extended by inheriting basic functionality from them, and redefining the unwanted portion.

The paper concludes that there is a need for the development of new strategies as these form the core of the framework. At present there is no strategy that addresses the state-based testing of classes.

A point that is not addressed by the paper, is the difficult task of creating complex objects that are to be passed as parameters. This must surely be one of the more difficult tasks that has to be performed. Another point is the construction of the strategies themselves, there is no description of how they actually function or interact with the framework.
5. Traditional Testing Techniques

It is felt that traditional testing techniques (see [14] for examples) are easily applicable to object-oriented programs. The data representation must be tested initially by state-based testing, then individual features can be exercised until their required coverage level is achieved.

State-based testing has a tendency to ignore the results that are returned by features. Therefore a degree of functional testing is required to validate the external view of the class. Various traditional functional techniques are available, and can be applied to the features of a class. As was stated before, the traditional testing process should be altered to:

This requires a small adaptation to the techniques. A feature of the class under test can be tested either in isolation or in unison with the other features of the class. If the features are exercised in unison, there is no need for extra code to generate the starting states for test cases, otherwise, extra code is required to generate the starting state of the object under test. The code used to generate the starting states during the state-based testing can be used for this purpose. Many different states should be used, as this is likely to have a definite effect on the coverage of any feature.

Figure 8. An adaptation of the Von-Neumann model of processing

C.D.Turner - 48 - 02/02/93
6. The MKTC suite of tools

This section will briefly outline the suite of tools that have been developed by the author to enable the evaluation of the state-based testing technique. As time has progressed, the tools have grown and increased in complexity. The tools are no longer for evaluation, but enable the state-based testing to be accomplished with easy, by removing the majority of the tedium associated with the repetitive generation of test cases. The suite comprises of three tools: MKTC (MaKe Test Cases), MKMFTC (MaKe MakeFile for Test Cases, and TESTRUN).

MKTC as the expanded name suggests, is used for the generation of the actual state-based test cases. It requires as input a test script file which describes to the tool the class under test and the tests that must be created. It will be described in slightly more detail below.

MKMFTC generates a Makefile\(^\text{24}\) which allows the automatic compilation and linking of all the test cases in the current directory. It not only generates a Makefile for the state-based test cases, but also for the 'ordinary', or traditional test cases. MKMFTC will be described in slightly more detail below.

TESTRUN allows the user to automatically execute all the test cases in the order that they were created. In a similar manner to MKMFTC, it not only deals with the state-based test cases, but also the 'ordinary' ones. TESTRUN allows the resultant output from all of the test cases to be logged to a file for perusal at a later date. Also, it is possible to generate a file with a description of all the test cases that failed (unless the fail causes a system crash). TESTRUN will also be described in slightly more detail below.

A more detailed description of the tools is available in a separate report [19].

\(^{24}\) A file that describes dependencies between the modules of a program. It also describes the command sequences that are required to build the final program, or programs.
6.1. MKTC

MKTC provides a number of features to the user:

- automatic generation of state-based test cases from the test script file
- a report of the test cases generated, listing the starting state, the finishing state, and details of the generators that were used to generate the initial state.
- an automatically generated class that allows the user to perform state-based tests either within their own test cases, or within the middle of the features of the class under test. It provides on the fly parsing and syntax checking of the state descriptions to be tested for.
- The ability to only replace the test cases within the current directory that have changed. This is useful when correcting errors in the test script file. If this facility is not used, then the Makefile will cause the automatic recompilation of all state-based test cases whether they have changed or not.
- The ability to simply perform a syntax check and minimal semantic analysis on the test script file without actually generating the test cases.
- The ability to provide a great deal of verbose debugging output reporting to the user exactly what is going on during the execution of MKTC. This is useful for debugging errors within the test script file.
- The ability to perform all the tasks involved in the complete test generation process without actually generating the test cases themselves. All the expansions of the test case descriptions are performed, the states are checked against the invariant, and the search for generators that could be used to create the initial states for the test cases. This is useful for the user-verification of the potential test cases that should be generated.
- The ability to report to the user a list of the generators that will be required by the test script file.

The test script file has seven sections. Some are optional, others are not. The first is the header information. It contains details such as the name of the class under tests, the test engineer's
name, the date, and any C++ include files that will be required for the correct compilation of the test cases.

The second section is used to communicate to MKTC the substates that are to be used during the current test run. For each substate, there is a simple declaration which allocates a unique number to it and describes the number of different substate values that it contain. Throughout the file the user is allowed to place comments enhancing the readability of the file and increasing the ease with which the file can be referred to at a later date.

The next section of the file describes to tool the data scenarios that will be used during the testing. It contains one declaration per scenario, informing the tool of the measurable state that the scenario is. For a more detailed description of data scenarios, see the section headed "Data Scenarios" on page 25.

The next section describes the invariant that is to be used by the tool to discard illegal test cases that might otherwise be generated. The tighter (the more clauses) the invariant, the greater the chance has of catching the unwanted test cases. Also it notifies the tool of the name of an added feature of the class under test which applies an invariant to the data-members of the class. This will be applied before and after each test case to enhance the discovery of failed test cases. This section is however, optional.

The next section contains a list of generators. A generator is a piece of code that places an object of the class under test into a specific state. The generators can be chained together to create the required state for a test case. Consequently, declared with each generator are the starting state of the object that is required for this generator, the finishing state that the generator will put the object into, and any code that is required to tidy up after the test cases. The extra code is used to return memory to the heap, close open files ...etc. It is possible to specify a generator's starting state as a scenario, therefore allowing the user precise control over the initial situation of the test case.
The next section simply informs the tool of the names of the extra features that were added to the class under test allowing the testing of substate values.

The final section describes the test cases. They are grouped under the name of the feature that is under test, however, a feature's name can occur as many times as is required, thus allowing the user to carefully order the test cases if required. The test consist of two parts, a mapping from the starting state to the expected finishing state, and a portion of code that executes the test. The starting state for the test can be specified as a scenario, which in turn requires a specific generator for that scenario. Also it is possible to declare multiple object that must be created and initialised for the test case. The user supplies the code, therefore any objects that are required as parameters can also be created by the user.

Both the starting state, and the expected finishing state can be expressed as a range of states, in which case, the tool will automatically generate all potential combinations of the supplied states. This means that a single declaration can result in the generation of a number of test cases, thus demonstrating the expressive power of the tool.

The starting state is used by the tool to locate a sequence of generators that create the required state. This removes the majority of the tedium in generating the test cases by automatically reusing the generators, and automatically surrounding the test portion of code with checks for the expected state at that point in the execution.

Only a brief outline of the facilities provided by MKTC have been provided here to demonstrate that the method outlined above has been evaluated, and has a degree of tool support already.

6.2. **MKMFTC**

This is a simple tool which requires a small file in the current directory which describes the modules that will be required for the test cases to compile. It scans the current directory
looking for test cases that match its expected file mask\textsuperscript{25}. These test cases are then incorporated into the Makefile with a description of how to combine the required modules to form executable test cases.

### 6.3. TESTRUN

This is also a simple tool which scans the current directory for any executable test cases that match its file mask. These are then executed, the output is sent to a file, and any failures logged to a separate file. The output file can then be scanned at a later date by the user as an aid to debugging why particular test cases failed.

\textsuperscript{25} A file mask is a description that contains 'holes' allowing the mask to match multiple file names.
7. Summary

This report has first described a method for testing object-oriented programs. This method includes a new technique for validating the correct interaction between the features of a class. It is this emphasis on the interaction and coupling between the features of a class that is prevalent in object-oriented programming. Also suggestions were provided for the adaptation of more traditional testing techniques to the object-oriented paradigm to provide a more complete view of the testing of object-oriented programs.

Also provided was a brief description of a suite of tools that enables the user to perform the state-based testing of classes more efficiently by automating much of the repetition involved.

At present, the state-based testing technique is still under going evaluation, and it is hoped that this will provide a much greater insight into the effectiveness of the technique on different types of classes. The classes that are component parts of the suite of tools are currently undergoing validation by a combination of state-based, functional, and structural testing techniques.

Reuse has been heralded as one of the major benefits of the object-oriented paradigm. Users are able to produce validated reusable components that can reduce the cost of software maintenance by reducing the number of potential errors that will require correction.
8. References


C.D.Turner - 56 - 02/02/93


### 9. Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Data Type (ADT)</td>
<td>A data type whose implementation is hidden. Its external behaviour is usually defined by a formal specification.</td>
</tr>
<tr>
<td>Acceptance testing</td>
<td>Testing of a system that is performed by the customer to make sure that it conforms to the specification that was laid down when the project was started.</td>
</tr>
<tr>
<td>Algebraic specification</td>
<td>One of a number of different methods for formally specifying a system, or component. It comprises a list of effects that are achieved when routines of an ADT are used in conjunction with each other, that is, it specifies a unit behaviour, by describing the interactions between the routines of the unit.</td>
</tr>
<tr>
<td>Assertion</td>
<td>A statement about the value of a variable or variables at a particular point in the execution of a program.</td>
</tr>
<tr>
<td>Child class</td>
<td>A class derived from another one via inheritance. Also known as a derived class.</td>
</tr>
<tr>
<td>Class</td>
<td>The unit of program construction is a (class-based) object-oriented language. A class is the static compile-time template from which objects are created.</td>
</tr>
<tr>
<td>Client</td>
<td>An object that calls another object. The object being called is said to be the supplier of services; the class that called the supplier is know as the client.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Constructor</td>
<td>The first routine of an object to be called. It is usually used to create the initial state of the object.</td>
</tr>
<tr>
<td>Control-flow</td>
<td>Control-flow is the path through the program that execution takes. It is affected by branches and conditional statements such as for and while loops.</td>
</tr>
<tr>
<td>Control-information</td>
<td>This is information that is not data, but is used to flag an event to another part of the program. It usually takes the form of Boolean variables.</td>
</tr>
<tr>
<td>Cyclomatic complexity measure</td>
<td>This is a metric for determining the complexity of a program. It is also a measure of how much of the cyclomatic complexity of a program has been executed by the current test suite.</td>
</tr>
<tr>
<td>Data-flow</td>
<td>This is the flow of data through a program. It is based upon the definition and uses of variables, and how they are affected by the decisions made during the programs execution.</td>
</tr>
<tr>
<td>Data-member</td>
<td>These are the variables that form the representation of an object. In Eiffel they are part of the group known as attributes. The name however, is taken from that used by the C++ community.</td>
</tr>
<tr>
<td>Data-representation</td>
<td>This is a collective name given to all of the data-members of a class together. Together, all the data-members of the class form the data-structure that is used to store and manipulate the concepts that the class will deal with.</td>
</tr>
</tbody>
</table>
Data-scenario - This is a particular situation, or situations that are specific to the data-structure of the class. There may not be any detectable difference between scenarios using the allocated substates for the class, this is the reason that this facility is used in addition to the normal state-based testing.

Debugging - The process of tracking down and removing errors from a program.

Derived class - A class that has been defined by inheriting from another class (known as the base class).

Destructor - The last feature of an object to be called. Its task is to release any resources that have been used by the object, and to generally tidyup after the object has deceased.

Dynamic data-structure - This is a data-structure that changes structure during the execution of the program. It usually consists of a series of nodes linked together in a particular way by pointers.

Exhaustive testing - Exhaustive functional testing is the process of running a program with every valid input value, and every invalid input value. In the vast majority of cases, it is infeasible. Exhaustive structural testing is the process of exercising every path through a program. This is infeasible for all but the simplest of programs.

Feature - This is the name given to a routine that is part of a class. Features are used to manipulate the values of the data-representation, or return the values to the caller, or to interact with the external world.
Formal specification - This is a specification which is based upon a mathematical statement about the expected functionality of a program. Formal specifications can then be used to produce the program directly, via steps known as data-reification. Each of these steps can be proved, therefore proving that the final program matches the initial specification. However, they require a great deal of expertise in the manipulation and proof of the specification.

Functional testing - This is the testing of a program, or component with respect to its expected functionality alone.

Functions - These are components of a program that return values to the caller.

Instantiation - This is the process of creating an object from its template (its class).

Integration testing - The testing process used for the combination of the separate units of a program. During integration testing, emphasis is placed on the interfaces between the units, rather than the code of the units. It is performed after unit testing.

Member function - Another name for a feature. Member function is the name used within the C++ community.

Method - Another name for a feature. Method is the name used within the Smalltalk community.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>The basis of object-oriented programming. It is a run-time unit which has a data representation, and a collection of features that interact with the representation.</td>
</tr>
<tr>
<td>Parent class</td>
<td>The opposite of a child class. A parent class is a class from which others are derived. It is also known as a base-class.</td>
</tr>
<tr>
<td>Partial state</td>
<td>This is the combinations of a number of the substates of an object. However, it is not the combination of all of the substates (the state).</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>It simply means &quot;many forms&quot;. It is used to describe the ability of an object created from a child class, to replace an object of the parent class. It allows objects to change type during the execution of a program.</td>
</tr>
<tr>
<td>Procedural programming language</td>
<td>A language which uses the procedure as the basic unit of programming construction.</td>
</tr>
<tr>
<td>Routine</td>
<td>A name for a segment of code that has a defined interface through which clients must interact. It is also another name for a feature.</td>
</tr>
<tr>
<td>State</td>
<td>This is the combined value of all of the substates that are part of the object.</td>
</tr>
<tr>
<td>State-based testing</td>
<td>The process of testing an object by exercising each feature with all possible states. This not only test the effects of the state upon the features responses, but also tests the effects of the features upon the state.</td>
</tr>
</tbody>
</table>
Structural testing - The process of generating test cases designed to exercise specific portions of the program, thus taking the programs structure into account.

Substate - A particular data-member of the object.

Substate-value - These are the values that can be stored by the substate. They are a set of specific and general values. Specific values are individual values that have a special significance. General values are groups of values that are all in the same manner by the features of the program.

System testing - The testing of the system as a whole, rather than as separate sections, or as individual units. It can include various different techniques including stress testing.

Test driver - An extra routine that is provided by the tester to generate the initial scenarios for test cases.

Test stub - An extra routine that is provided by the tester, to imitate another part of the system. Test stubs and drivers are used to test routines in isolation, by replacing everything around the routine under test.

Testing resources - These are expendable items which are available to a project when performing testing. They include the time of personnel, the number of personnel, and the amount of processor time available.

Unit testing - The process of testing the individual units of a program in isolation. Usually test stubs and drivers are used to achieve the isolation.
| Validation | The testing of the software at the end of the development effort to ensure that it meets its requirements [9]. That is, does the software do what it is supposed to do. |
| Verification | The evaluation of the software during each phase of its life-cycle to ensure that it meets the requirements laid down in the previous phase [9]. That is, "Have we built it correctly?" |