Testing Object-Oriented Software: a Survey

ROBERT V. BINDER
RBSC Corporation, 3 First National Plaza, Suite 1400, Chicago, IL 60602-4205, U.S.A.

SUMMARY

Research and practitioner literature on testing object-oriented software published up to the end of 1994 is summarized. The contribution of each source to eight topics is presented: (1) abstract data type verification and testing as it relates to object-oriented testing; (2) testing theory—fault hypotheses for object-oriented software and adequate testing (several fault taxonomies are presented); (3) automatic model validation—techniques and tools for testing executable object-oriented representations; (4) test case design—heuristic and formal techniques to develop test cases from object-oriented representations and implementations; (5) testability—factors in controllability and observability; (6) test automation—assertions, state manipulation, comparators, object identity and built-in tests; (7) test process—strategies to organize and manage the activity of testing object-oriented implementations; and (8) experience reports. Appendices provide several cross-references.

KEY WORDS abstract data types; built-in test; C++; CLOS; design for testability; Eiffel; fault taxonomy; formal methods; formal verification; integration testing; model validation; object-oriented analysis; object-oriented design; object-oriented testing; object-oriented programming; Objective-C; oracles, regression testing; Smalltalk; software process; test coverage; test adequacy; test case generation; test automation; unit testing

1. INTRODUCTION

This paper is a survey of research and practitioner work on testing object-oriented software published up to the end of December 1994. A concerted effort has been made to locate all available sources. It is a certainty that literature in this area will grow. The author welcomes additional citations. Readers are encouraged to study the primary sources, since many details are necessarily not reported here. Other less extensive surveys are available (Binder, 1994a and 1994c; Hayes, 1994; Overbeck, 1993 and 1994a; Turner and Robson, 1993c); however, all sources discussed in them are included here. No attempt is made to cover the extensive literature on related software testing issues. Several general texts on software testing are available, including those of Myers (1979), Howden (1987) and Beizer (1990). An early survey of verification, validation and testing techniques is that by Adrion et al. (1982). This introduction considers basic issues, the context of testing object-oriented software, and provides an overview of the survey.
1.1. Perceptions of testing in object-oriented development

Software testing is necessary to produce highly reliable systems, since static verification techniques cannot detect all software faults. While only a small proportion of developers practice adequate testing (Hetzel and Gelperin, 1991), few conventional developers would contend testing is irrelevant. Current perspectives on object-oriented testing can be characterized as optimistic, minimalist or expansive.

The optimistic view is that the technology and software processes typically associated with object-oriented development obviate or greatly reduce the need for testing. Iterative and incremental development using a ‘robust’ class library is seen as a process not unlike writing source code with an incremental compiler. Once a new class is syntactically correct and has been smoothly integrated (to the developer’s satisfaction), development is done. Subsequent changes are viewed as refinements to be conducted in the same manner. Formal testing (the development and systematic execution of a test plan and test cases) is not necessary. After recommending ‘... each class must be thoroughly tested’, Taylor (1992) speaks to the optimist. ‘To many object programmers, this degree of analysis and testing may seem like a major intrusion on what should be a quick, intuitive process of class definition.’ Many popular object-oriented methodologists do not discuss testing at all (Coad and Yourdon, 1990; Shlaer and Mellor, 1992; Embley et al., 1992; Martin and Odell, 1992; Coad and Nicola, 1993).

The minimalist view is that testing remains necessary for object-oriented implementations, but it will be easier and less costly. Booch (1991, p. 212) argues ‘... the use of object-oriented design doesn’t change any basic testing principles; what does change is the granularity of the units tested’. A text on advanced C++ offers a single sentence on testing which pronounces, ‘Black-box testing proceeds as for any system.’ (Coplien, 1992, p. 209). The observation of Rumbaugh et al. (1991) characterizes this point of view:

‘Both testing and maintenance are simplified by an object-oriented approach, but the traditional methods used in these phases are not significantly altered. However, an object-oriented approach produces a clean, well-understood design that is easier to test, maintain, and extend than a non-object-oriented design because the object classes provide a natural unit of modularity.’ (Rumbaugh et al., 1991, p. 144).

A similar perspective is offered by Wirfs-Brock et al. (1990) ‘... object-oriented design means that the entities of the system can be isolated and tested one at a time. ... Entities can be shown to function before being plugged into the system. ... places where a responsibility was omitted in the design or made part of the wrong entity can be spotted and filled.’ (Wirfs-Brock et al., 1990, p. 11).

The expansive point of view is that typical features of object-oriented systems require new approaches to attain adequate testing, beneficial effects of class modularity notwithstanding. New design strategies and processes for testing under a formal testing regime are needed. Siegel (1992) notes that ‘Many people are suggesting that testing needs and costs will be lower for OOD systems. In reality, this probably will not be true.’ In the first careful analysis of object-oriented testing, Perry and Kaiser (1990) report:

‘... we have uncovered a flaw in the general wisdom about object-oriented languages—that “proven” (that is well-understood, well-tested, and well-used) classes can be reused as superclasses without retesting the inherited code. ... [Testing] may still turn out to be easier ... but there are certain pitfalls that must be avoided.’
In addition to inheritance and polymorphism, Smith and Robson (1990) argue that other typical features of object-oriented languages are complex and therefore error-prone. With large class libraries, it may be difficult for a developer to comprehend the intended usage. Activation by message passing is argued to be significantly different from conventional activation; data flow involves objects instead of primitive types. Genericity and abstract super classes present similar problems. It may be possible to instantiate a generic class with an inappropriate type. If a generic class is changed, then all instances of it should be retested. Testing for reusability should be performed to certify reusability.

While ‘object oriented software validation can be based on conventional approaches, [this] is not enough. There remains a large field to be studied.’ In particular, consistence in the abstraction hierarchy resulting from inheritance and correct behaviour of concurrently executing objects are identified as significant problems (Tamai, 1991).

Object-oriented development will experience ‘different types and proportions of errors that require a different approach to testing’ compared to conventional development methodologies and languages (Firesmith, 1992 and 1993b). Encapsulation reduces but does not eliminate the hazard of global data faults (intra-class access can cause the same problems). Path testing is limited to methods and must consider exception handling and concurrency. Since methods are typically small, the path selection problem is often trivial. Boundary value analysis is ‘of limited use’ owing to the beneficial effects of strong typing and ‘proper’ data abstraction. Message/parameter equivalence classes are relevant instead of variable equivalence classes.

Encapsulation can pose an obstacle to testing (Berard, 1993). Since the state of an object can only be reported by a method on the object, a partially tested method may have to be relied upon for reporting the state. Parametrized classes, specialization and name-clashes under multiple inheritance pose unique testing problems.

1.2. Issues in testing object-oriented software

Although much of what is known about testing conventional systems applies, object-oriented development presents unique testing problems which require new solutions. While the sources surveyed here do not always agree on approach, nearly all agree that there are significant differences. Four main concerns are evident.

(1) Fault hypothesis. Are some facets of object-oriented software more likely than others to contain faults?
   (a) What kind of faults are likely?
   (b) How can revealing test cases be constructed?
   (c) What effect does the scope of a test (method, class, cluster, subsystem, system) have on the chance of revealing a fault?
   (d) To what extent are fault models specific to applications, programming languages and development environments? What is general and what is not? For example, are strategies that are useful with statically typed languages ineffective with dynamically typed languages?

(2) Test case design. What models are useful for reasoning about probable sources of error? What techniques may be used to construct test cases from models, specifications and implementations?
(a) How much testing is sufficient for methods, classes, class clusters, application systems and components of reusable libraries?
(b) To what extent should inherited features be retested? Under what conditions must features inherited from a superclass be retested in a subclass?
(c) How should abstract and generic classes be tested?
(d) What models for understanding collaboration patterns are useful for testing?
(e) How can models of the abstract and concrete state of an object, class or system under test be used?
(f) How can state-dependent behaviour be verified when state control is often distributed over an entire object-oriented application?
(g) How can clients and servers linked by dynamic binding (polymorphism) be tested?
(h) How can collection and container classes be tested?
(i) How can regression tests be formulated and conducted to ensure the integrity of classes modified under the open–closed principle (Meyer, 1988)?
(j) How can information in representations and implementations be used to develop or automatically produce the expected results for a test case?

(3) Test automation issues.
(a) How should test cases be represented?
(b) How should test cases be applied? What is the proper role of stubs and drivers?
(c) How can test results be represented and evaluated?
(d) When can reports about the state of an untested object be trusted?
(e) What is an effective strategy for integration testing?
(f) What role can built-in tests play?
(g) Should built-in tests be provided by application components, programming languages and/or development tools?

(4) Test process issues.
(a) When should testing begin?
(b) Who should perform testing?
(c) How can testing techniques help in preventing errors?
(d) How are testing activities integrated into the software process model?
(e) How can testing facilitate reuse?
(f) To what extent should reusable test suites be considered a necessary part of a reusable component library?
(g) What is an effective test and integration strategy given the iterative approach preferred for object-oriented development?

Many answers have been posed for these questions. For some issues there is consensus; with others, there is debate. Some have yet to be examined in detail.

1.3. Survey overview

At present, literature on testing object-oriented programs is limited when compared to the extensive material on software testing in general. (As of first quarter 1991, Beizer (1991) estimated over 2000 journal articles and 25 books had been published on conven-
tional testing). However, interest and work in this area is growing, as Figure 1 and Table I show. The surveyed sources address a wide range of practical, theoretical, technical and process subjects. The following categories are used to present these concepts.

(1) Abstract data type verification and testing. Abstract data types are more like objects than not. Several approaches to object-oriented testing have been guided by work on the verification of abstract data types (ADTs). Sources are reported here to set the context for the primary subject of this survey. This section is not a complete survey of ADT verification.

(2) Testing theory. Theories of testing provide a model which identifies likely faults and facilitates production of test cases. Explicit and implicit fault hypotheses, coverage criteria, and several fault taxonomies are presented.

(3) Model validation. Several testable (executable) representations of object-oriented systems have been developed. Verification capabilities of formal (mathematically rigorous) and semi-formal systems are presented.

(4) Test case design. Many heuristic and formal techniques for test case development have been proposed. Representation-based and implementation-based approaches are presented by the scope of the implementation under test: a class or cluster of classes, integration of clusters and subsystems, and testing of entire application systems. These techniques correspond to the familiar unit, integration and system test categories.

(5) Testability. Technical and process strategies are presented which facilitate testing.

(6) Test automation. Reports about instrumentation, test tools and built-in tests are presented.

(7) Test processes. Process proposals which outline a general plan of attack for testing in the context of object-oriented development are presented.

(8) Experience reports. Reports about how testing has been conducted in real-world projects are presented.

Nearly all of the sources deal with theory, implementation and process issues to some extent. To avoid arbitrary classification, sources are discussed by subject. Appendices provide a list of the sources by subject, programming language and development method.

Concepts are presented chronologically. When a concept has appeared several times (e.g. conference proceedings followed by a journal), chronology follows first publication, but content follows the most recent source. To avoid frequent repetition of phrases like ‘Jones (1999) argues that testing reduces defects’ or ‘it is argued that testing reduces defects’, the summary of the work of Jones (1999) will read: ‘testing reduces defects’. This is to be understood as the source’s point of view, not as this survey author’s assertion or commentary. The summaries are intended to be neutral précis in so far as is possible. Any interpretations and conclusions are those of the author of this survey and have not been reviewed by the originating authors. Long quotes are followed by a page number.

1.4. Terms

An object-oriented programming language provides explicit support for encapsulation, object identity, set abstraction, inheritance and self-recursion (Wegner, 1987; Stroustrup,
Programming languages which are not object-oriented are referred to as conventional.

Summaries follow each source’s usage for object-oriented terms. Otherwise, Smalltalk usage is generally followed: class, sub-class, super-class, method, message, selector, protocol, object, instance variable, etc. Features of a class include private instance variables and methods as well as its public interface. A client class uses a method defined in another class or itself; a server class responds to a client message. Class flattening makes the effects of inheritance explicit by source code expansion (Meyer, 1988). A flattened class shows all the features it inherits.

Generally terms follow applicable IEEE/ANSI standards (IEEE, 1986 and 1990). A failure is an observable deviation from a required capability: missing or incorrect output, unacceptable performance in time or space, or abnormal termination. A failure results from a fault, which is an incorrect or missing software component(s). Fault and defect are synonymous. A fault results from human error. The implementation being tested is referred to as the implementation under test (IUT); thus method under test (MUT), object under test (OUT), class under test (CUT), system under test (SUT), etc. An equivalence class refers to a subset whose members are expected to agree on some characteristic. This is not ‘class’ in the object-oriented programming language sense.

### 2. ABSTRACT DATA TYPE TESTING AND VERIFICATION

#### 2.1. Introduction

A complete discussion of object-oriented testing requires consideration of abstract data type (ADT) verification and testing techniques. Many ADT concepts and features have
been incorporated in object-oriented specification and programming. Lessons learned from ADT development can be applied to object-oriented development. Some object-oriented testing approaches make explicit use of ADT techniques (Smith and Robson, 1992; Doong and Frankl, 1994; Parrish et al., 1993b). The purpose of this section is to provide a context for understanding. It is not intended as an exhaustive discussion of ADT issues.

2.1.1. Basic concepts

An abstract data type is a software module that encapsulates data and operations. Encapsulated data is accessed only by the operations 'exported' from the ADT interface. Operations on a 'type' constrain the mapping between input and output parameters in an operation. An operation which reports state without changing it is a function; a procedure may change state. ADTs were developed as a vehicle to explore formal verification and proof of correctness (Guttag et al., 1978). With a formal specification for an ADT, it can be proved (manually) that an operation will produce the specified result. ADTs have been studied extensively; some representative reports include those of Liskov and Zilles (1975), Guttag et al. (1985), Liskov and Guttag (1986) and Weide et al. (1991).

There are two primary techniques for ADT specification. An algebraic specification has two main parts: operations and axioms. An operation is specified by its interface (name and arguments). Axioms are equations which define the behaviour of an operation in terms of other operations. This constitutes a kind of 'algebra', hence the name. Data structures may be elemental or composite and must be representable in set theoretic terms. Operations provide the syntax of the type; axioms the semantics. Model-based specification defines individual operations by pre- and post-conditions, or by a non-procedural expression of input/output mapping. Model-based specification provides an explicit definition for each operation; algebraic specification defines operations by reference. Algebraic specification makes the relationship among operations explicit; model-based specification does not. Algebraic specification is compact and is well-suited to automatic transformation. On the other hand, model-based specification is argued to be easier to understand and more scalable. The Larch specification language (Guttag et al., 1985) uses both:

'[Larch] shows the relationship between the two approaches to data abstraction: algebraic specifications are used to define the abstract types and their operations. The interface specifications are based on the abstract models approach, because they contain invariants, abstract I–O assertions of the operations being implemented, and a function that maps stored values into abstract values'. (Ernst et al., 1994, p. 290).

Figure 2 summarizes both specification styles.

Several programming languages have been designed to support ADTs, notably Ada and

<table>
<thead>
<tr>
<th>Algebraic Specification</th>
<th>Model-Based Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>= functions</td>
<td>= operations</td>
</tr>
<tr>
<td>+ domain conditions</td>
<td>+ interface</td>
</tr>
<tr>
<td>+ axioms on functions</td>
<td>+ pre-conditions</td>
</tr>
<tr>
<td></td>
<td>+ post-conditions</td>
</tr>
<tr>
<td></td>
<td>+ invariants</td>
</tr>
</tbody>
</table>

Figure 2. Approaches to ADT specification
Modula-2. Research languages include Alphard, CLU, Concurrent Pascal, Euclid, Gypsy, Mesa and Russell (Shaw, 1984). While algebraic ADTs have influenced the design of object-oriented languages (Stroustrup, 1992), there is no direct implementation of axioms in popular object-oriented programming languages. The Eiffel language provides an object-oriented implementation of model-based specifications (Meyer, 1992). The require and ensure clauses of Eiffel can approximate axioms but are neither required nor as powerful (Weide et al., 1991; Meyer, 1992). The assert macro in C++ and Objective-C can be used in a similar manner.

2.1.2. Relation to object-oriented testing

Object-oriented programming languages share many features with abstract data type (ADT) specification and programming languages. Both rely on encapsulation of implementation. Both 'abstract' these contents by providing an interface consisting of parametrized operations and preventing any other kind of access. Both support information-hiding and effective modularity. There are significant differences, however. Classes are typically constructed by inheritance; ADTs do not provide inheritance. A class hierarchy with dynamic binding provides method polymorphism in most object-oriented languages; ADTs are typically not polymorphic. ADTs were developed to research formal verification; object-oriented languages generally were not concerned with this. ‘I designed C++ to solve a problem, not to prove a point, and it grew to serve its users.’ (Stroustrup, 1992, p. 24) There are many formal ADT specification techniques. A formal specification can be proven correct and algorithmically transformed into an implementation. The correspondence of the implementation and such specifications may also be proven. Many object-oriented languages are inimical to this kind of transformation (Olthoff, 1986; Leavens, 1991).

2.1.3. ADT verification and testing

ADT verification and testing can focus on one of several possible relationships among software artifacts. (This model is an extension of the verification scheme in the work of Weide et al., 1991). These artifacts are as follows.

(a) Informal or semi-formal requirements.
(b) Formal abstract components; for example, a Z or Larch specification.
(c) Meta-model for abstract components; for example, Z syntax.
(d) Concrete components; for example, a Modula-2 implementation.
(e) The observable behaviour of a concrete component.

Verification, validation and various forms of testing are each concerned with different relationships among these artifacts.

(i) Model validation attempts to establish confidence in the sufficiency of a formal abstract component with respect to its informal behavioural requirements.
(ii) Formal verification attempts to prove a concrete component is correct with respect to an abstract component.
(iii) Consistency checking evaluates the consistency of an abstract component with a meta-model of the representation technique.
(iv) **Black-box testing** evaluates the conformance of the observable behaviour to the abstract component.

(v) **White-box testing** evaluates observable behaviour with respect to the concrete component.

(vi) **Product validation** evaluates conformance of observable behaviour to requirements.

Each technique can make a unique contribution to defect elimination. Figure 3 depicts the techniques and corresponding relationships. The ADT techniques surveyed here are presented according to these categories.

The theory and structure necessary for formal verification is also useful for testing. Formal verification of ADTs has been extensively discussed and is not considered here, except for sources which have a direct relation to object-oriented testing.

ADT testing approaches have focused on finding fault-revealing operation sequences, automatic test generation and automatic evaluation of tests. Many ADT test approaches exploit the algebraic equality relationship for automatic test case generation and test evaluation. In effect, the specification provides a kind of built-in automatic oracle (Gannon

---

**Figure 3. ADT artifacts and verification techniques**
et al., 1981; Dauchy et al., 1993; Doong and Frankl, 1994). Relationships among axioms can be exploited to generate test sequences and test case data (Bougé et al., 1986; Choquet, 1986; Gaudel and Marre, 1988; Bernot et al., 1991; Dauchy et al., 1993; Doong and Frankl, 1994).

Most ADT test approaches generate operation sequences which are hoped to be fault-revealing (Jalote and Caballero, 1988; Jalote, 1992; Zweben et al., 1992; Parrish et al., 1993b; Doong and Frankl, 1994). Tests using model-based specifications select sequences by analysing define–use relationships among exported operations (Zweben et al., 1992; Parrish et al., 1993b). A coverage measure may be derived from such a test suite. In contrast, tests using algebraic specifications make use of constraints on operations (axioms) to select paths by a substitution process.

Automated support for model validation and consistency checking is discussed in Section 4. Product validation is not discussed here as it can be accomplished by established methods which are not specific to ADTs or object-oriented development.

2.2. Formal verification

2.2.1. Prover’s assistant

An early presentation of the ADT approach argues that algebraic specification facilitates proofs by factoring complexity and allowing automated support. ‘Writing the axioms in a certain style allows them to be used as reduction rules so that proofs become largely symbol manipulation exercises.’ (Guttag et al., 1978, p. 1062).

‘Testing’ uses a direct implementation input to the ‘data type verification system’ (DTVS). A direct implementation for data type \( T \) is an ‘implementation whose representation part is a subset of the syntactic specification of \( T \) and whose program part is a subset of the semantic specification of \( T \).’ (Guttag et al., 1978, p. 1059).

The verification process has three steps: (1) input the specification and implementation to DTVS; (2) use DTVS to generate axioms and equality axioms for the specification (verification conditions); (3) the user attempts to prove the verification conditions, using an automated support for substitution of previously established results. This process is reported to be useful for discovering missing assumptions. The tool keeps track of proofs and does some syntactic checking.

2.2.2. ModPascal

The ModPascal research system was developed to study the application of ADT concepts to object-oriented programming languages (Olothoff, 1986). A primary goal of this research was to identify programming language features necessary to allow formal ADT verification to be used in conjunction with object-oriented implementation. Four extensions to Pascal are needed.

(1) Module type objects, providing the type interface definition. This defines exported operations and encapsulated features.

(2) Enrichment objects to support type mappings.

(3) Instantiation objects to support specialization.
(4) Instantiation objects to support declarations that are resolved by type substitution (genericity).

This research demonstrated that a language specified with denotational semantics can provide a basis for proving equivalence between a formal specification and its implementation.

'In our view, the main point is that a uniform and consistent embedding of an object-oriented language in [an algebraic] verification context . . . is only possible if the relevant concepts of ADT theory have associated counterparts in the programming language. Also, the underlaid [sic] semantics have to be compatible i.e. algebraically based to avoid the otherwise necessary transformations of one semantic model into another.' (Olthoff, 1986, p. 437).

This approach allows an algebraic specification and its implementation to be checked 'on an algebraic level' for correct implementation. This verification can be performed on a single operation (method), but not on the entire ADT (class).

2.2.3. Modular type/subtype verification

Formal verification of ADT specifications can be facilitated by requiring consistency between supertypes and subtypes (Leavens, 1991). The specification and verification techniques are 'modular' in that they can be applied to one type at a time using an extension of Larch. The extension allows message polymorphism. The type of a parameter in an operation is specified as a base supertype, but may be bound to any subtype associated with a base type. Consistency between super and subtype specifications is concerned with this extension. The goal of modular verification is to obtain subtype verification by proving the supertype correct.

Each parameter in an operation belongs to a type hierarchy and may take on any subtype. This signature determines which subtype operation is to be bound to a particular message. The verification problem is to assure that any type which can be bound to an operation is consistent with that operation, and to do so by relying on verification of the root type's specification.

To rely on supertypes for verification, 'you must impose strong constraints on the design of subtypes of a given type.' Syntactic constraints are: (1) a subtype must accept all messages accepted by its supertypes; and (2) the types resulting from operation evaluation must be a monotonic hierarchy—i.e. the result of an operation on the subtype must be a subtype of the result for the corresponding operation on the supertype. Semantic constraints are used to show that for each subtype pre- and post-condition the super's pre- and post-conditions also hold. The constraints are added to the specification in the form of a relation called a simulation. A simulation is a set of mappings between the abstract values of the supertypes and subtypes.

'... the process of designing a subtype involves designing part of your program's correctness arguments at the same time. You use the idea of a simulation relation to ensure that the new type will be a subtype of the desired existing types.' (Leavens, 1991, p. 80).
The paper demonstrates a method for modular verification of algebraic specifications where types support message polymorphism. Many of the issues considered are relevant for corresponding problems in object-oriented testing.

2.3. Black-box testing

2.3.1. The DAISTS system

The DAISTS research system is a self-testing programming language for abstract data types (Gannon et al., 1981). It supports ADTs based on the Simula class. It uses axiomatic specifications to generate test drivers that perform both activation and evaluation.

There are four steps in the test process: (1) prepare axioms (specifications); (2) prepare a class (an implementation); (3) prepare named instances of the ADT under test to be used for test cases (‘test points’); (4) prepare ‘testsets’ to exercise a particular axiom. A testset is either named constants or named test points.

The test strategy generates two call sequences, executes them, and compares the results. One sequence is obtained from the left-hand side of an axiom, the other from the right-hand side of the same axiom. If the implementation is correct, the results of both sequences should be equal. For example, suppose X and S are type variables, x and s are instances, and the following stack axiom is present:

\[ \text{top}(\text{push}(\text{X}, \text{S})) = \text{X} \]

This axiom requires that after \text{push} has placed some X on the stack, the \text{top} operation returns the same X. To verify the axiom, DAISTS generates a test driver as follows:

```plaintext
proc stack_test(in x, stack s)
    if top(push(x, s)) = x
        put "pass"
    else
        put "fail"
endif
```

The left and right sides of the axiom are instantiated and compared for equality. If they are unequal, a fault has been revealed.

The ADT under test must implement an equality operator to compare the implementation under test with a testpoint. The DAISTS processor provides automatic instrumentation, giving code statement coverage and axiom expression coverage. It generates operations which should give equivalent results and uses the equality test to determine pass/fail. Thus no oracle (for output values) is needed. The equality test allows automatic test evaluation, but cannot detect all kinds of fault.

DAISTS was viewed as an effective tool for revealing faults, especially those related to boundary conditions. ‘The most surprising thing about using DAISTS is that, even for the simplest examples, it is impossible to second-guess the system responses: the bookkeep-
ing required to see that a set of tests succeeds, including the structural requirements, can only be done accurately by a computer.’ (Gannon et al., 1981; pp. 220–221). An experience report on the use of DAISTS appears in a paper by McMullin and Gannon (1982).

The logic programming approaches, LOFT, and ASTOOT systems (discussed below) have all adapted this evaluate-and-compare strategy.

### 2.3.2. Black-box testing with CLU

A textbook on programming in CLU (an ADT implementation language) argues ‘A good way to generate black-box test data is to explore alternate paths through the specification. These paths can be through both the ‘requires’ [pre-condition] and ‘effects’ clauses [post-condition].’ (Liskov and Guttag, 1986, p. 163). If either clause has an and, or operation, test cases should be devised to exercise all conditions (false-false, false-true, true-false, true-true).

The test data should cause every possible signal to be raised. Tests for boundary conditions are suggested: typical values and all combinations of the largest and smallest allowable values. For example, string and array type operations are to be exercised on instances which are empty, contain one member and are full. Aliasing errors can occur when ‘A single mutable object is bound to two different formals.’ If a signature will accept two or more of the same argument type, then it is worth trying the case where both arguments are the same variable.

### 2.3.3. Logic programming for test data generation

Prolog can be used to generate test cases for ADTs with algebraic specifications. This approach takes advantage of the fact that ADT axioms may be readily represented in Prolog as rules. Prolog evaluates queries by searching for all values which make the statements in a given program true. Test data may be generated by representing axioms as Prolog statements and searching for combinations of values which satisfy all the rules. For example, a Prolog representation of an algebraic specification is shown in Figure 4.

Testing attempts to demonstrate that the implementation conforms to each axiom for the selected test points. This is done by instantiating both the axiom and the IUT with the same test values and comparing the results (the evaluate-and-compare strategy). The testing is conformance-directed in that no tests are generated for illegal operations.

The logic programming test data generators rely on the regularity and uniformity hypotheses (Weyuker and Ostrand, 1980; Bougé et al., 1986). Regularity assumes that a program behaves regularly for every value of some parameter of interest, e.g. the number of iterations in a loop. The number of tests is reduced significantly with this assumption, e.g. it is not necessary to try every possible value of a loop control parameter.

Uniformity assumes that domains are uniform, i.e. there are no subsets in the domain which will result in abstractly different behaviour, which implies that picking any arbitrary set of domain inputs will not result in a significant omission.

Uniformity may not hold if conditional axioms are present. The presence of conditional axioms complicates test data selection since each conditional introduces an additional path to consider. (For example, the axiom for an is_member operation on an integer set returns true if the argument is a member; false otherwise). This is essentially the same problem that occurs in path sensitization: the condition sets must be identified for a given path,
// Algebraic specification for an integer queue

IntegerQueue
operations
emptyq: () -> queue;
append: queue*int -> queue;
remove: queue -> queue;
first: queue -> int;
isempty:queue -> bool;

precondition
pre(first,Q) = (isempty(Q) = false)

axioms
A1:isempty(emptyq) = true;
A2:isempty(append(Q,I)) = false;
A3:remove(emptyq) = emptyq;
A4:isempty(Q) = true => remove(append(Q,I)) = emptyq;
A5:isempty(Q) = false => remove(append(Q,I)) = append(remove(Q,I),I);
A6:isempty(Q) = true => first(append(Q,I)) = I;
A7:isempty(Q) = false => first(append(Q,I)) = first(Q);

/* Prolog database that represents axioms of integer queue ADT */
/*A1*/
isempty(emptyq, true).
/*A2*/
isempty(append(Q,I),false).
/*A3*/
remove(emptyq,emptyq).
/*A4*/
remove(append(Q,I),emptyq) :-
isempty(Q,true).
/*A5*/
remove(append(Q,I),append(Q',I)) :-
isempty(Q,false),
remove(Q,Q').
/*A6*/
first(append(Q,I),I) :-
isempty(append(Q,I),false),
isempty(Q,true).
/*A7*/
first(append(Q,I),J) :-
isempty(append(Q,I),false),
isempty(Q,false),
first(Q,J).

Figure 4. Logic programs from algebraic specification (adapted from the work of Bougé et al., 1986)

test data must be selected, and infeasible paths must be dealt with. The number of operations for a type is referred to as its complexity measure.

The finite decomposition hypothesis states that a conditional axiom can be transformed into a set of equational axioms whose domains correspond to the original condition sets (Bougé et al., 1986). Conditions may be factored by rendering them in disjunctive normal form (Gaudel and Marre, 1988).

Although regularity and uniformity allow reductions, too many test cases were produced (Bougé et al., 1986). Two strategies were developed to limit the number of tests generated (Choquet, 1986). Preconditions were added to conditional axioms and translated to Prolog constraints. The search returned symbolic values (unevaluated expressions) for parts of the axiom.

2.3.4. LOFT system

The LOFT system (Logic for Functions and Testing) generates black-box ADT test cases. It extends earlier logic programming approaches with refinements to produce a practically small but effective set of test cases. A detailed theoretical presentation is given in the paper of Bernot et al. (1991). LOFT was used to validate an automatic control
specification for a subway train door controller and to test a subsequent implementation (Dauchy et al., 1993). The model validation report follows in Section 4.

LOFT differs from earlier logic programming approaches in two main ways: assurance of observability and user-controlled test case generation instead of automatic enumeration.

The use of the evaluate-and-compare strategy necessitates that all operations be observable and comparable. Thus, only observable terms are allowed; axioms with non-observable terms are automatically recast in an observable form.

Test data may be seeded by the user or randomly selected from the 'observable exhaustive test data set.' Test data selection uses equation resolution to identify values for the axiom domains. Test data generation is initiated by random selection of initial values. The user may also designate specific initial values.

Automatic test case generation is replaced by selection. The system supports 'theoretically justified selection strategies' (uniformity and regularity as interpreted for ADT testing). Positive conditional axioms are used for conditional narrowing, as directed by the user. The narrowing command set is shown in Table II.

Unfolding of conditional terms is accomplished by substitution of equivalent expressions. Since the unfolding may lead to a large number of tests, the user may designate limits for unfolding. Identification of search constraints requires careful analysis and an understanding of the application. A specific analysis is described that leads to a tractable test set for the subway train door controller. In this application, any one of a large number of alarms (conditions reported by sensors in the car) would be sufficient to cause an emergency stop. Thus for \( n \) alarms, there are \( 2^n \) emergency states. In this case, unfolding was limited by a user-specified `do_not_unfold` declarative, which terminates the Prolog search path.

‘Depending on the required safety level, less extensive decompositions can be considered sufficient. They can be obtained in two ways. We can add the `do_not_unfold` declarative on boolean operations in order to stop the decomposition sooner, but then the reached uniformity subdomains will not differentiate dangerous circumstances; or we can rewrite the specification a little, in order to test dangerous circumstances separately.’ (Dauchy et al., 1993, p. 241).

<table>
<thead>
<tr>
<th>Narrowing command</th>
<th>Effect on test case generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>Stop when equations match <code>do_not_unfold</code></td>
</tr>
<tr>
<td>Random choice</td>
<td>Select axiom randomly (not in lexical order)</td>
</tr>
<tr>
<td>Factorize</td>
<td>Share common occurrences of a term</td>
</tr>
<tr>
<td>Rewrite</td>
<td>Rewrite after each expansion</td>
</tr>
<tr>
<td>Eager</td>
<td>Discard right-hand terms which do not reference left-hand variables</td>
</tr>
<tr>
<td>Max depth</td>
<td>Limits the depth of an expansion</td>
</tr>
<tr>
<td>Init-inter</td>
<td>Limits the breadth of the expansion search tree</td>
</tr>
</tbody>
</table>
2.3.5. ADTs and ASTOOT

The ASTOOT system is based on ADT specification and testing concepts (Doong and Frankl, 1994). ADT testing is difficult, owing to complex input and output types. It is argued that an ADT implementation is correct if it is 'homomorphic to the algebra of equivalence classes. Intuitively, each equivalence class corresponds to a possible "abstract state" of the ADT.' This implies two conditions for a correct implementation: (1) the individual results of any two equivalent sequences of operations are the same ('indistinguishable'); and (2) the individual results of any two non-equivalent sequences of operations are not the same ('distinguishable'). The ASTOOT approach is discussed in Section 5.

2.3.6. SITE for implementations

The SITE system generates tests for ADT implementations from a reference implementation and test suite, i.e. a set of procedure operation sequences (Jalote, 1992). The details of the SITE system are discussed in Section 4, 'Model validation'. The test suite is executed against both the IUT and the reference implementation. The results from the IUT and the reference implementation are input to a comparator. If there are differences in the outputs, a fault has been detected. Figure 5 shows a block diagram for this approach.

2.3.7. Specification-based control flow and data flow testing

ADT tests can be derived from model-based specifications using a dataflow strategy (Zweben et al., 1992). Heuristic techniques to derive test cases are problematic in that the adequacy of the resulting test set is ambiguous. Since a package of operations has more interactions than its components in isolation, some systematic approach to exercising the entire package is warranted. A model-based ADT specification can be interpreted to
identify sequential constraints on operations, allowing identification of define–use
sequences.

Each operation is modelled as a node in a flowgraph, in the same way that a program
segment would be modelled in a testing flowgraph. Edges are identified by analysis of
pre-conditions and post-conditions for all operations. Operation sequences are identified
in a pairwise manner. For any operation \(A\), if the pre-condition of any other operation
(including \(A\)) can be satisfied by \(A\)’s post-conditions, then a feasible edge is drawn from
\(A\) to this operation, \(A'\). The edge means \(A'\) can be activated after \(A\).

To develop dataflow tests, definition and use nodes need to be identified for the type.
This is accomplished by analysis of post-conditions. Table III shows the rules for
determining definition and use.

The approach was found to be useful and compares favourably with the fault-revealing
power of corresponding program-based criteria. Several limitations are noted. Any of the
criteria can be satisfied with minimal type instantiations. For example, only four elements
are needed in a two-way list example. Some faults require a longer, repetitive sequence
of operations to be revealed. Identification of such sequences is not addressed by
this technique.

The authors conclude that conventional white-box testing techniques can be adapted to
testing ADTs from specifications. Analysis of coverage ‘suggests that strategies at least
as powerful as “all-uses” are necessary . . . somewhat higher-order members of families
of testing strategies such as “all-branches” and “all-uses” may be required in order to
guarantee that real defects in ADT implementations are revealed.’

2.3.8. Z predicate substitution

Tests can be generated from model-based ADT specifications (Z) in a system that
incorporates an automated test oracle and produces a specification coverage metric (Jia,
1993). The approach is to establish conformance of the implementation to the specification
by substituting values generated from the IUT into specification predicates. If this instantiation
is inconsistent with the specification, an implementation fault has been revealed.

A Z specification consists of a data schema and an operation schema. Z is an
abstract representation using the mathematical nomenclature of sets and functions. An

<table>
<thead>
<tr>
<th>Operation post-condition cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>old x is coded</td>
</tr>
<tr>
<td>nochange x or (&lt;x = \text{old } x&gt;) is coded</td>
</tr>
<tr>
<td>At least one reference to x is coded.</td>
</tr>
</tbody>
</table>

Define–use relationship for variable x

\(x\) is USED in the operation  
\(x\) is DEFINED in the operation  
\(x\) is NOT USED in the operation

\(\checkmark\) requires the exit value of a variable to be identical with its entry value.

\(\text{Note: The keyword old refers to the value of a variable at entry to the operation; nochange}\)

\(\text{requires the exit value of a variable to be identical with its entry value.}\)
implementation is concrete, defined over a concrete domain. Domain elements (or a class of elements) are states which may or may not be consistent with the type invariant. Specification-based testing requires effective mapping between these domains.

Concrete and abstract domains are mapped by data retrieve relations, expressed in ZED/RL. They 'indicate how abstract states can be constructed from concrete states.' In a correct implementation, 'each concrete state should represent a unique abstract state.' Concrete operations are modelled by expressions of the form:

\[(\text{output, resultant state}) = \text{operation (input, pre-state)}\]

There are two sufficient conditions for conformance:

1. for an initialization operation,

   \[\text{initial state} = \text{initialization operation (null state)} \text{ such that the invariant holds}\]

2. for all other operations,

   \[(\text{output, result}) = \text{operation (input, pre-state)} \text{ such that the abstract op-spec holds and the abstract invariant holds}\]

A test case consists of the name of an implemented operation and values for its parameters. Complete specification coverage would require testing 'all possible invocation sequences.' But this is 'infeasible' since there are an infinite number of such sequences. Two heuristics are offered to construct a 'reasonable' test suite: (1) make each after-condition and data condition in each schema true at least once; (2) for container types, every element must participate in every operation at least once. Test cases are manually prepared based on these heuristics.

Testing is accomplished in three steps with the specification, implementation, data retrieve functions and test cases in hand. First, a test driver is generated. The specification compiler reads the Z specification and retrieve functions. It produces source code for schema evaluators, abstractors and a test driver. A schema evaluator compares a concrete value with a specification domain and reports whether or not it is a member of that domain. The evaluator makes use of the ZED/LIB library of C++ classes that correspond to Z expressions. An abstractor is generated from the data retrieve functions and ZED/LIB classes.

Second, the driver and the IUT are compiled together to produce a harnessed implementation.

Third, the harnessed implementation reads the test cases, tracks predicates exercised, and reports pass/fail results. The test driver repeatedly performs the following tasks.

1. Read a test case.
2. Invoke an operation, saving before/after state and parameter values.
3. Construct the resultant abstract values using the abstractors.
4. Use the appropriate evaluators to check abstract values with respect to specification predicates and invariants.

A Z coverage metric is presented. It requires a kind of predicate normalization that
results in exclusive use of the disjunctive or operator (skolemization). This format allows an unambiguous combinatorial evaluation of predicate truth values, e.g. with two predicates: (false, true) and (true, false).

The specification-based nature of this approach provides several beneficial results. The oracle checks specification conformance. Coverage is defined in terms of specification predicates (as opposed to graph-based measures). Scalability is obtained as a result of using model-based specifications. Specification-based testing is argued to be more effective than manually developed assertions.

2.4. White-box testing

2.4.1. Assertions from Z

Implementations derived from proven specifications still need to be tested (Hayes, 1986). Automated proof is not always feasible. Manually produced proofs and code are subject to error and omission. The test strategy is to code data invariants and pre-conditions from proven specifications. White-box testing techniques are used to exercise ‘thoroughly’ ‘all parts’ of the IUT.

Z specification components can be used in several ways. The data type invariant can be used for checking the consistency of the state between operations. Pre-conditions can distinguish errors in the IUT from those in the test program or any client. Relations on states and individual operation input–output relations can be used for testing correctness of operations. Examples of Pascal code are presented which implement Z invariants and pre-conditions.

Invariants can reveal faults that might otherwise be masked. However, ‘the implementation could be disastrously wrong and still maintain the invariant.’ Invariant checking may impose too high a performance penalty, so it is typically removed from the delivered system. Pre-condition checking may be useful for debugging a test driver and subsequent clients. It is advisable to leave these checks in the delivered system if possible.

‘This is an essential requirement for public interfaces such as operating system calls or widely used packages; it can help sort out debates about which component is at fault.’ (Hayes, 1986, p. 128).

To check the I–O relation, two programs should be developed from the same specification and tested with the same inputs. If the output differs, a fault has been revealed (which could reside in either or both programs). Identification and programming of loop invariants may be difficult. They may be effective in debugging a loop that resists correction by other means. Testing ‘does not exclude the possibility of latent errors . . . [which shows] the inherent weakness of program testing compared to program verification.’ (Hayes, 1986, p. 129).

In discussing this approach, Jia (1993) comments that:

‘Checking the validity of the assertions on the concrete states of an implementation is not equivalent to checking the validity of the predicates in the specification, unless the assertions are consistent with the specification. It is not a trivial task to assure consistency between the assertion and the specification. The testing based on assertions means little when the assertions are inconsistent with the specification.’ (Jia, 1993, p. 366).
2.4.2. **White-box testing with CLU**

A textbook on programming in CLU (an ADT implementation language) argues that white-box testing is needed to reveal components that may be missed by black-box testing (Liskov and Guttag, 1986). Examination of the code is necessary to decide the appropriate number of iterations for loops: fixed loops should get at least two iterations and all-ways terminations. Variable loops should iterate zero, one and two iterations and all terminations. Recursive functions should try arguments that result in zero or one recursions. For iterators, attempt zero, one or two productions.

2.4.3. **Operation pairing**

Pairs of operands are automatically selected and executed, providing a test suite which is ‘...strictly program-based and therefore does not require formal module specifications.’ (Parrish et al., 1993a, p. 50). This is argued to be appropriate when operations are added to an existing ADT a few at a time. In this situation, rapid integration of a new operation is desirable; specification-based testing is characterized as ‘problematic’ since it requires a complete set of operations. This may pose an obstacle to integrating a partially complete ADT. The effort (cost) to prepare the necessary formal specification and devise a test plan may be perceived as excessive. ‘Programmers may be unlikely to engage in a testing activity that substantially interferes with progress in development.’ (Parrish et al., 1993a, p. 49).

The test approach is straightforward. For each operation \( x \), one test case is developed that will cause \( x \) to be followed by every other operation, including \( x \) itself. For an implementation with \( n \) operations, there are \( n^2 \) such operation pairs in the test set. These sequences can be automatically generated. Such a test set provides at least ‘all-edges’ coverage. If each operation is marked as ‘in’, ‘in–out’, or ‘out’, then a test suite of pairs that covers ‘all-uses’ can be produced and executed in \( O(n^2) \) time.

This approach (in contrast to that of Zweben et al., 1992) does not consider the feasibility of any given operation pair. All edges (operation pairs) are automatically generated, so the approach ‘is only applicable to defensive modules.’ To be defensive, the IUT must respond gracefully to all illegal sequences, that is, no corruption of state will occur if, for example, push is attempted on a full stack.

An implementation of this approach for Ada programs is described. It allows selection of operation subsets for test generation, and will exercise any set of operation pairs. A state reporting operation must be added to the IUT. After running the test suite, the user decides pass/fail by examining the output of a reporter operation.

3. **TESTING THEORY**

3.1. **Overview**

This section presents some conceptual foundations for testing object-oriented systems. It is not a comprehensive treatment of testing theory. The purpose is to provide a context by summarizing sources cited in papers on technique or process.

Two basic issues frame all testing techniques: ‘What is likely to go wrong?’ and ‘To what extent should testing be performed?’ Since the number of possible tests is infinite for practical purposes, rational testing must be based on a fault hypothesis—an assumption about where faults are likely to be found. This hypothesis suggests a test case design
model. The design model yields a test suite when applied to a representation or implementation. Assessing coverage of the modelled features typically requires instrumentation of the implementation under test.

The object-oriented programming paradigm has several unique features: encapsulation, inheritance, and polymorphism (Goldberg and Smith, 1989; Wegner, 1987; Stroustrup, 1988). These features are at once beneficial and hazardous, leading to fault hypotheses for object-oriented software. This section develops the concept of a fault hypothesis, summarizes fault hypotheses based on inheritance, polymorphism, state and the C++ language. Perspectives on adequacy and coverage are discussed. Several fault taxonomies are summarized.

3.2. The role of a fault hypothesis

A fault hypothesis is an essential part of a testing approach. It is an assumption based on common sense, experience, suspicion, analysis or experiment about the relative likelihood of faults in some particular aspect of a system under test. A fault hypothesis answers a simple question about a test technique: ‘Why do the features called out by the technique warrant test effort?’.

There are two general fault hypotheses which correspond to two basic testing strategies: (1) conformance-directed testing, which seeks to establish conformance to requirements or specifications; and (2) fault-directed testing, which seeks to reveal implementation faults.

Conformance-directed testing proceeds by constructing tests which are sufficiently representative of the essential features of the system under test, applying the test suite and deciding if the response is appropriate. Conformance testing relies on a non-specific fault hypothesis: any fault will do to prevent conformance.

The sufficiency of the testing model with respect to system requirements is crucial to establish conformance. Conformance-oriented testing need not consider potential implementation faults in detail, but must establish that a test suite is sufficiently representative of the requirements for a system. Conformance sufficiency must be defined with respect to a particular scheme for representation.

Fault-directed testing is motivated by the observation that conformance can be demonstrated for an implementation that contains faults. Searching for faults is a practical and prudent alternative to conformance (Myers, 1979). Since the combinations of input, state, output and paths are astronomically large, efficient probing of an implementation requires a specific fault hypothesis to direct the search for faults.

Either a convincing argument or strong evidence that a particular kind of probing has a relatively good chance of revealing a fault is needed. There are several kinds of specific fault hypothesis: (1) suspicion (Hamlet and Taylor, 1990); (2) extrapolation from past experience; (3) an assumption that a particular kind of fault is often associated with a small number of circumstances (e.g. evaluation of boundary values in decision segments) so that, by trying all such circumstances, all faults of this type will be found; and (4) an argument (or evidence) about the kinds of error that are likely to be made and the kind of fault they cause. For example, the data flow fault hypothesis is based on a suspicion:

‘Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed.’ (Rapps and Weyuker, 1985, p. 367).
The functional testing fault hypothesis is based on a correlation of faults and functions:

‘Studies of program faults revealed that they are often associated with embedded subfunctions, or special functional aspects of a piece of code. The studies indicated that if these subfunctions had been considered separately in the construction of test cases then the chance of discovering faults would have risen dramatically.’ (Howden et al., 1993, p. 3).

Conformance-oriented techniques should be feature sufficient: they should at least exercise all specified features. Fault-oriented techniques should be fault efficient: they should have a (very) high probability of revealing a fault. A well-formed fault hypothesis should explain why relative sufficiency or efficiency obtains for its associated testing technique.

3.3. Fault hypothesis: inheritance

inheritance is an essential part of the object-oriented paradigm (Wegner, 1987; Stroustrup, 1988). It may be used (abused) in many ways.

‘... a concrete component’s implementation must understand the implementation details and subtle representational conventions of all its ancestors in order to implement that component correctly. Unless care is taken, it is possible to introduce components that seem to work properly yet, by manipulating their ancestor’s internal data representations, violate subtle and implicit conditions that the ancestors require for correctness.’ (Weide et al., 1991, p. 51).

Inheritance can contribute to errors in several ways: (1) it weakens encapsulation; (2) fault hazards are introduced which are similar to those associated with global data in conventional languages; (3) it can be used as an unusually powerful macro substitution for programmer convenience, as a model of hierarchy (either problem or implementation) or as a participant in an implicit control mechanism for dynamic binding. It is not unusual to see all three purposes at work in a single class hierarchy. This overloading can lead to undesired side-effects, inconsistencies, and incorrect application behaviour. Deep and wide inheritance hierarchies (lattices under multiple inheritance) can defy comprehension, leading to errors and reduction of testability.

3.3.1. Forgotten methods

Where reuse is accomplished through many levels of inheritance, proper usage of heavily reused features may become obscured. This creates an opportunity for error in Objective-C. ‘... longstanding bugs have persisted because nobody thought to verify that deeply inherited methods, particularly easily forgotten object-level methods like copy and isEqual were overridden to adapt to the peculiarities of the subclasses.’ (Cox, 1988, p. 46).

3.3.2. Incorrect initialization

Inheritance in Objective-C can make it difficult to understand source code (Taenzer et al., 1989). A subclass at the bottom of a deep hierarchy may have only one or two lines of code, but may inherit hundreds of features. Without the aid of a class flattener, the interaction of these inherited features is difficult to understand. In Objective-C, all superclass instance variables are visible in subclasses. This poses fault hazards similar to
unrestricted access to global data in conventional languages (this problem may be avoided by using accessor/modifier methods instead of statement level access). Initialization can easily go awry. Objects are created by the new method. New is often inherited and uses a class-specific initialize method to actually set subclass instance variable values. Determining how initialize is used in a subclass requires examination of the superclass that defines new. The initialize message must be sent to super, not self. Now, suppose new is refined, and does not send initialize to self. Super’s initialize will not execute, causing the inherited behaviour to be incorrect. The compounding effect of polymorphism is discussed below.

3.3.3. Semantic mismatch

Specialization with simple inheritance (overriding or renaming) means that subclass methods have different semantics and require different tests (Smith and Robson, 1990). Multiple inheritance allows superclass features with the same names, providing the opportunity for misnaming. Changes to either superclass will affect the subclass. Repeated inheritance occurs when a subclass derives features from the same superclass more than once, and is argued to offer still more subtle opportunities for misnaming and semantic mismatching.

3.3.4. Test reduction

An optimistic view of inheritance argues that it will reduce the testing necessary in derived classes, provided that the base classes are ‘thoroughly’ tested (Cheatham and Mellinger, 1990). Message passing and inheritance influence how much testing will be needed. With unaltered inheritance, ‘little additional testing [of a derived class] is needed. At most the interface should be retested.’ An overridden or unique method ‘... must be tested as a new member function.’

3.3.5. Testing axioms

A formal analysis of adequate testing of object-oriented programs is presented by Perry and Kaiser (1990). Requirements for adequate testing are framed in terms of Weyuker’s (1988) software testing axioms. It is shown that intuitive conclusions about test adequacy may be wrong. For example, owing to inheritance and dynamic binding, re-testing of both superclasses and subclasses is necessary when a superclass method is changed. The implication of four axioms is considered.

(1) Adequate testing of programs that have the same function but different implementation requires different test suites (antiextensionality axiom).
(2) Adequate testing of programs that have the same shape (i.e. flowgraph) but different data domains requires a different test suite (general multiple change axiom).
(3) Adequate testing of a program invoked or used by different callers requires a different test suite for each such context (antidecomposition axiom).
(4) Tests which are individually adequate for units of a program are not collectively adequate for the assembled units (anticomposition axiom).

Encapsulation does not obviate the need for integration testing. When a new or changed
unit is integrated, even with encapsulation, it is necessary to test all other components that depend on it.

(a) When a class is changed, the changed class and the classes that explicitly depend on it should be (re)tested.
(b) When a superclass is changed, the changed class and all its subclasses should be (re)tested.
(c) When a subclass is changed or added, all inherited methods should be (re)tested in the context of the subclass. The new subclass provides a new context for these methods, which has not been tested.

Overriding is typically used to provide superclass/subclass interface consistency while allowing a different subclass implementation. An overriding method in a subclass can be implemented by a different algorithm, different functionality, or both. The test suite for the overridden method will almost certainly not be adequate for the overriding method.

Multiple inheritance ‘... unfortunately cause[s] very small syntactic changes to have very large semantic consequences.’ Suppose Z is a subclass of classes X and Y, and method m is present in both superclasses. Z originally used X.m but is changed to use Y.m. Z must be retested and it is likely that the test suite for Y.m will not be appropriate for the new Z.m. Thus any change that involves the binding precedence rules will probably require retesting of superclass methods in the subclass context with different test data.

Changes to a subclass can conflict with unchanged inherited features or unmask superclass faults. Retesting of these unchanged features in the new context is therefore warranted. An exception to this rule occurs with a ‘pure extension’ subclass where there are ‘... new instance variables and new methods and there are no interactions in either direction between the new instance variables and methods and any inherited instance variables and methods.’

3.3.6. Inheritance and correctness

If an implementation is obtained by inheritance, the specification-based test suite for inherited features may be reusable, reducing the test effort for a subclass (Doong and Frankl, 1991). However, inheritance may easily be used as code-copying convenience, independent of any relationship between the superclass and subclass specifications.

‘... we cannot expect the correctness of [a subclass] to have any relation at all to the correctness of [the superclass] unless there is a relationship between the specifications of the two classes. It is often, but not always, the case that the specification of [the subclass] is a specialization or extension of the specification of [the superclass].’ (Doong and Frankl, 1991, p. 173).

With the superclass tested, it is necessary to test the subclass against both its own specification and that of its superclass. The subclass is expected to respond to inherited messages as does the superclass, but with its own data state. Thus, the implementation of the superclass may suggest ways a subclass can fail. Consider a subclass derived from a linked list. The relative value of items for insert and delete operations are unlikely to be related to faults in the linked list. But suppose the superclass is an ordered tree. The relative order matters for correct operations, and suggests a testing strategy.
3.3.7. Class versus type

Inheritance can implement type/subtype relationships, but this requires care. Irregular type hierarchies will lead to incorrect polymorphic message bindings. A '... subtype relationship is a relationship between specifications, while a subclass relationship is a relationship between implementation modules.' (Leavens, 1991).

A class hierarchy may represent a network of problem domain relationships or share implementation features by some kind of factoring scheme (Purchase and Winder, 1991). Attempting both in a single class hierarchy is a source of bugs. With multiple inheritance, an incoming selector may match two or more methods from different superclasses, resulting in an incorrect binding. Multiple inheritance can also result in unanticipated interaction among inherited features. Abstract classes cannot be tested without an implementation; each implementation requires separate testing. It is relatively more difficult to develop, understand and use complex, deeply nested class structures.

3.3.8. Classless language

In a classless object-oriented language, changes under inheritance need only minimal regression and integration testing (Trausan-Matu et al., 1991). Objects are like new records in a DBMS. 'The problem of instances rises only in class-based languages.' In class-less languages, 'everything is an object... which can be used as a prototype for the definition of other objects. When new objects are added, it may or may not result in changes to the old hierarchy structure. The new objects should be tested as a system on their own.' Although dynamic inheritance may be supported, it is seen as untestable. 'Dynamic change of existing inheritance links in an ordinary OO program may have deep consequences, and so it is not usually allowed.'

3.3.9. Name clash detection

An analysis of faults possible with multiple inheritance is presented by Chung and Lee (1992, 1994). Multiple and repeated inheritance can result in 'name confliction' or 'gene-confliction' (name clash) when the same name is used for any feature of two or more superclasses of the class under test. Chung and Lee present a theoretical basis for name clash detection. Specific techniques for test case design and execution are not discussed. An automatable name clash detection approach is needed since these faults are argued to be hard to find. An algorithm based on a graph model of multiple inheritance is developed. This provides a way to detect non-singular inheritance.

'It is difficult to find the repeated inheritances, much less test those implicit software errors. Furthermore, the problem of finding out all the repeated inheritances is an NP-complete problem... Since it is impractical to exercise all repeated inheritances in an inheritance graph, techniques to guide the testing units become important. The idea is that repeated inheritances of an inheritance graph is composed of a set of unit repeated inheritances (URIs) and name-confliction errors could be found and solved easily from them.' (Chung and Lee, 1992, p. 381).

Some causes of name clashes are considered. With multiple inheritance, a class inherits (directly) from two or more parent classes that contain features with the same names. For example, D is derived from classes B and C. Repeated inheritance occurs when a common
superclass is present in a multiple inheritance network. For example classes B and C are derived from superclass A, and class D is derived from B and C.

The graph model of inheritance represents each inheritance path as a sub-graph. This model is used to analyse inheritance (multiple or repeated). Several algorithms are presented to trace inheritance paths and identify name clashes. The algorithm enumerates pairs of inheritance paths for a given derived class. It is shown that an inheritance graph with an enclosed region necessarily contains a repeated inheritance. No specific procedure or criterion is offered for exercising a class under test to reveal name clash errors. The technique seems to suggest that one should attempt execution of all repeated inheritance pairs.

3.3.10. Class dependencies

Within a C++ class or class cluster, dependencies may arise due to side-effects introduced by inheritance or global data (Thuy, 1992). Access to protected or private features should be limited and controlled.

'Since most of the problems raised by the object-oriented approach are due to method redefinition, it must be kept under close control. The legal [legitimate] reasons for method redefinition are optimization, extension and definition of deferred methods.' (Thuy, 1992).

3.3.11. Sub/super class coupling

'We have at least two reasons for an inherited operation not to function in a descendant: [1] if the descendant class modifies instance variables which the inherited operation assumes certain values for, [2] if operations in the ancestor invoke operations in the descendant.' (Jacobson et al., 1992, p. 321).

3.3.12. Testing axioms reinterpreted

Four guidelines are offered by Berard (1993) to establish the scope of testing: (1) similar functions may require different tests. An adequate black-box test suite for a base function may not be adequate when the function is inherited and overridden; (2) an adequate test suite for a base function may not be adequate when a function is bound by precedence ordering under multiple inheritance; (3) different contexts may require different tests. An adequate test suite for a base function may not be adequate when the function is inherited in a different context; (4) class testing alone is not always sufficient—integration testing is usually required. For example, suppose a function interface remains the same when the method implementation is changed. The uses and inherited methods as well as the method should be retested. These guidelines are derived from the testing adequacy axioms of Weyuker (1988) and their interpretation by Perry and Kaiser (1990).

3.3.13. Multiple inheritance

Multiple inheritance presents many error possibilities (Moreland, 1994). Cycles are possible and differ among C++ compilers. Problems include multiple addresses for a single object, aliasing and inadvertent 'self' assignments. If members are not explicitly qualified by type, name clashes can occur. The semantics of virtual functions change. Pure virtual functions may be renamed and redefined. Additional concerns include effects
of public and private inheritance, abstract classes versus concrete classes and the visibility of superclass data members.

3.4. Fault hypothesis: polymorphism

Polymorphism replaces explicit compile-time binding and static type-checking by implicit runtime binding and runtime type checking. The semantics, syntax, and mechanisms implementing it differ in each programming language. Although polymorphism can facilitate compact, elegant, and extensible code, problematic aspects are identified by many sources (Smith and Robson, 1990; Purchase and Winder, 1991; Thuy, 1992; Wilde and Huitt, 1992; Jacobson et al., 1992; Meyer, 1992; Wilke, 1993; Jüttner et al., 1994a and 1994b; McCabe and Watson, 1994; McCabe et al., 1994a; Ponder and Bush, 1994; Moreland, 1994). However, Jacobson et al. (1992) assert that proper use can reduce testing under certain conditions.

‘If you add a descendant class [in a polymorphic server hierarchy] and test this class on its own, you do not need to test [its] client classes again, so long as you have used inheritance for sub-typing.’ (Jacobson et al., 1992, p. 319).


3.4.1. The ‘yo-yo’ problem

An analysis of reusability suggests many fault hazards due to the message binding mechanisms in Objective-C (Taenzer et al., 1989). As classes grow deeper and application systems grow wider, the likelihood of misusing a dynamically bound method increases. A class implements a message if it defines a method for the message that does not use inherited behaviour; it refines a method if it defines a method for a message and then sends the message to super.

‘Often we get the feeling of riding a yo-yo when we try to understand one of these message trees. This is because in Objective-C and Smalltalk the object self remains the same during the execution of a message. Every time a method sends itself a message, the interpretation of that message is evaluated from the standpoint of the original class . . . This is like a yo-yo going down to the bottom of its string. If the original class does not implement a method for the message, the hierarchy is searched (going up the superclass chain) looking for a class that does implement the message. This is like the yo-yo going back up. Super messages also cause evaluation to go back up the hierarchy.’ (Taenzer et al., 1989, p. 33).

An example of this problem is given using five classes as outlined in Table IV. C1 is a superclass, C2 is its subclass, etc. In this example, the implementation of method A uses B and C; B uses D. Messages to these methods are bound according to the class hierarchy and the use of self and super. ‘The combination of polymorphism and method refinement (methods which use inherited behaviour) make it very difficult to understand the behaviour of the lower level classes and how they work.’ (Taenzer et al., 1989, p. 33).

Figure 6 shows the difference between lexical message structure and a dynamic trace that can result from it. Suppose an object of class C5 accepts message A. C5’s A is inherited from C4, so the search for A begins at C4.A. C4.A is a refinement, so C3.A
Table IV. The ‘yo-yo’ problem

<table>
<thead>
<tr>
<th>Class</th>
<th>Method A</th>
<th>Method B</th>
<th>Method C</th>
<th>Method D</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Implements Sends self to B and C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td>Implements Sends self to D</td>
<td>Implements Sends self to C</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>Refines Sends super to A</td>
<td>Refines Sends super to B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>Refines Sends super to A</td>
<td></td>
<td>Refines Sends super to A</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td></td>
<td></td>
<td></td>
<td>Refines Sends super to A</td>
</tr>
</tbody>
</table>

is checked for an implementation. C3.A likewise refers to C1.A, where the implementation is found and executed. C1.A now sends message B to itself, causing B to be bound to self (an object of class C5) so the search for B begins back at C5. An implementation of B is found in C3. C3.B sends B to super (C2). C2.B is executed, sending D, which is again bound to self (C5). The search for D continues up to C2, where D is implemented. C2.D is executed for the C5 object, which then sends message C to self (still C5). The search for a C5.C is resolved at C4.C, which sends it to super (C3). Since C3 does not implement method C, C2.C is checked. The implementation C2.C is executed on self (an object of class C5).

A trace of some actual code with an eight level class hierarchy is presented. One message begins at the bottom and passes up and down through 58 methods before coming to rest.

3.4.2. Polymorphism considered harmful

Object polymorphism with late binding can easily result in messages being sent to the wrong class. It may be difficult to identify and exercise all such bindings. Meyer (1992) characterizes the problem as follows:

‘What is to prevent a [polymorphic] redeclaration from producing an effect that is incompatible with the semantics of the original version—fooling clients in a particularly bad way, especially in the context of dynamic binding? Nothing of course.’ (Meyer, 1992, p. 47).

Sub-classing is not necessarily sub-typing, so dynamic binding which uses an irregular class hierarchy, can produce undesirable results. But errors are still possible even with well-formed subtypes and formal specification. Suppose superclass method \(x\) has been verified. Later a subclass overrides method \(x\). The correctness of subclass \(x\) is not guaranteed because its pre- or post-conditions may not be the same as those of superclass \(x\). ‘Even if the pre-conditions and post-conditions were textually identical, the assertions might have different meanings for each type [class].’ (Leavens, 1991, p. 78).
Dynamic binding shifts responsibility for correct operation to the server at runtime. Thus a client may assume or require a method not provided by the server, may misuse an available method, or may incorrectly construct an interface signature.

'It is obvious that the scope for subtle errors with polymorphism is enormous. A debugging agent, unaware of how the + method is overloaded, may have false confidence in a bugged variant after other incarnations pass prescribed tests. Complex polymorphic relations can

---

*Figure 6. Trace of 'yo-yo' execution (adapted from the work of Taenzer et al., 1989)*
serve to confuse the implementor, debugging agent and maintainer alike by, for example, obscuring the point of origin of the bug.' (Purchase and Winder, 1991, p. 16).

An experience report on work to rehost a Smalltalk compiler calls this a kind of 'Catch-22 . . . Operations performed are determined [at runtime] from the variable types, and the variable types are deduced [at runtime] from the operations.' (Ponder and Bush, 1994). Methods are typically small (a few lines of code). The same method name is often used by many classes. Understandability suffers.

'Such systems are also sensitive to subtle naming errors. In a dynamically searched type [class] hierarchy there is a large pool of available procedure names, increasing the chances that a mistyped procedure call will invoke the wrong procedure.' (Ponder and Bush, 1994, p. 36).

Several test strategies for revealing faults due to polymorphism are discussed in Section 5.

3.5. Fault hypothesis: state-related

State-based testing is concerned with detecting value corruption or sequentially incorrect message responses. A state-based fault hypothesis suggests how state corruption or incorrect behaviour can occur. State-based testing strategies are discussed in Section 5.

3.5.1. Equivalent sequences

Certain activation sequences of class methods may be expected (specified) to yield identical results. In this situation, the inability of a pair of objects under test to produce equivalent results is a failure. The fault hypothesis here is non-specific: many kinds of faults could cause this kind of failure. However, the approach is conceptually simple and can be readily automated.

The ASTOOT approach is based on 'checking whether sequences of operations which, according to the specification, should yield the same state, actually do.' (Frankl, 1989). Member functions which should support associative evaluation are fault-prone ' . . . defects have been discovered by applying associativity rules to member functions. That is, if string s1 is null, and string s2 is not null, s1 < s2 should yield the same results as s2 < s1.' (Fiedler, 1989). It is suggested that an automated 'search' of the class under test could yield 'identity' sequences (Smith and Robson, 1992). An identity is an arbitrarily long sequence of messages which should result in no net change to an object. The test compares the initial and final state of the object under test for equality. The mechanics of the search or the test are not described.

3.5.2. Data scenarios

The interaction between the features and the state of an object are 'central to object-oriented programming' (Turner and Robson, 1993d). The implementation of complex data structures are to be analysed to determine 'which particular changes can occur to the structure, and when they can occur.' This analysis produces a data scenario, which is a state model for a single data structure. Data structures that provide intra-class communi-
cation by persistent state are fault prone. 'The more features of a class interact via the
representation, the more effective state-based testing is likely to be.' States are identified
by partitioning data member domains.

Data scenario testing combines conformance and failure directed testing. The goal is to
demonstrate that transitions result in a correct state or remain (correctly) in the same
state. The approach seeks to reveal faults causing a transition to an undefined state, the
wrong state or incorrectly remaining in a state.

3.5.3. Implications of cooperative design

Cooperative client/server classes increase the likelihood of state control errors (Binder,
1994b). When used as a server, a cooperative class assumes the client will not issue
incorrect message sequences or send messages with incorrect content (Meyer, 1992). Servers rely on clients to implement some or all of the server's state control. In contrast,
defensive design makes the server responsible for correct operations and enforcing correct
usage of itself under all circumstances. A defensive server is thus tolerant of client faults.

In contrast to conventional languages, preservation of state over successive activations
is an essential feature of object-oriented programming. State control must be implemented
by message sequences. With a cooperative design, state control is a shared responsibility,
requiring correct implementation of a single concept in two or more separate software
components. If the server is heavily used, there will be many clients which each must
correctly implement the server's state control rules. Binder concludes state control faults
will be predominant in object-oriented systems for several reasons.

(a) Overall control in object systems is implemented by many interfaces among small
components. The resulting 'delocalization of plans' (Soloway et al., 1988) reduces
intellectual tractability, increasing the likelihood of errors.
(b) Cooperative control is a popular implementation pattern, increasing the opportunities
for control errors in proportion to its use.
(c) Cooperative control is inherently complex, increasing the likelihood of control
errors when it is used.

For example, a client of an object sends a message putting the object into state $x$. Message $p$
is illegal in state $x$, but a different client could easily send message $p$ while
the server is in state $x$.

3.5.4. Observability of state faults

Functional or structural tests applied to individual member functions '... are not
adequate for detecting errors due to interaction between member functions through an
object state.' (Kung et al., 1994). For example, suppose a flag $f$ is incorrectly set in
member function $a$. Member function $b$ uses $f$ to determine an output. It is argued that
decision coverage of $a$ and $b$ alone would not reveal this fault, since the incorrect output
from $b$ only occurs after a particular sequence of $a$ and $b$. However, modelling the class
as a state machine and deriving an $n$-switch cover (Chow, 1978) reveals this fault.
3.5.5. Non-specific state faults

Many of the state-based testing approaches discussed in Section 5 are conformance-oriented and therefore rely on a non-specific fault hypothesis. No analysis is offered in these approaches of the specific kind of errors or faults likely to cause the failures which demonstrate non-conformance. They simply seek to demonstrate that the IUT conforms to its specification. The ASTOOT approach makes use of the fact that a faulty implementation may not produce the same state after accepting functionally equivalent but operationally different sequences (Doong and Frankl, 1994). Similarly, McGregor and Dyer (1993a, b) and Hoffman and Strooper (1993a) adapt (partially) Chow's (1978) state-based conformance technique, which is known to reveal certain kinds of control fault. The OOSE methodology calls for a transition cover of class state models (Jacobson et al., 1992).

3.6. Fault hypothesis: C++

Many aspects of the C++ language are error-prone. C++ inherits some deficiencies of C; understanding the behaviour of arrays, constructor/destructors, and virtual functions requires consideration of complex special cases and counter-intuitive behaviour (Sakkinen, 1988). The object under test may be used as input parameter to itself. Suppose s1 is a string. Then s1.Append (s1) may reveal pointer faults or insufficient isolation of self-referents (Fiedler, 1989). The use of assertions in base classes, constructors and destructors is recommended to reveal memory allocation and pointer corruption faults (Thielen, 1992). ‘C++ is a powerful language that facilitates writing clear, concise, efficient and reliable programs. However, its many features can be bewildering to the uninitiated, and injudiciously used, they can lead to programs that are confusing, bloated, slow and bug-ridden.’ (Shopiro, 1993, p. 211). ‘Most C++ bugs are due to unsafe features of the language: pointer problems, memory problems, uninitialized values, improper casts, improper union usage.’ (Reed, 1993). Incorrectly duplicated instance variable names (pointer aliases) are easily-made errors which are difficult to detect. Aliasing occurs when ‘...an object, or part of it becomes globally accessible. As a result, it is possible to change the state of an object without calling any of its routines.’ (D’Souza and LeBlanc, 1994, p. 34).

A list of 50 C++ 'gotchas' provides a detailed catalogue of easily made errors (Cargill, 1993). These errors typically result when the developer fails to consider the implication of some subtle C++ semantics. For example, in the following code fragment, only Base::~Base is called when Derived is deleted. Since Derived's destructor is not activated, its allocated memory becomes garbage (this can be prevented by declaring the base destructor as virtual).

```cpp
// Wrong destructor called, scoping error
class Base {
public:
    ~Base () {
        // ...
    }
};

class Derived : public Base {
public:
    ~Derived () {
        // ...
    }
};
```
A simple class extension can lead to bugs which are hard to find. If a member function
\texttt{Derived:clear} is added in the second example, \texttt{Derived} class objects will no longer use
the global \texttt{clear}, possibly resulting in a failure (Reed, 1994a).

//Wrong member function called, scoping error
class \texttt{Base} {
    public:
        \texttt{clear}();
    
    class \texttt{Derived}:public \texttt{Base} {
        public:
            \texttt{member\_function}();
    
        \texttt{clear}(); // calls Global clear
    
Several authors provide lists of common C++ errors.

Firesmith (1993b): (1) violation of encapsulation due to friend functions, misuse of private,
protected and public scope, and incorrect type casts; (2) references in a base class to a
derived class; (3) pointers, arrays, unions, and casts (type conversions) which defeat type
safe usage; (4) omission of virtual declarations in polymorphic superclasses; (5) ambiguities
in overloading public and private members; (6) assignment is not always initialization;
(7) overloading ignores returned types; (8) incorrect object references (size and address).

Davis (1994): (1) absence of input parameter range checking; (2) using an incorrect ‘this’
pointer; (3) using an incorrect object pointer for a virtual class; (4) state corruption due
to errors in constructor, destructor or invalid array references.

Moreland (1994): (1) local names can hide global names; (2) overloading and defaulting
are similar in appearance but have very different semantics; (3) implicit type coercion
with overloaded operators can produce strange results.
3.7. Adequacy and coverage

An adequacy criterion defines a set of software components to be exercised (e.g. all statements executed at least once). The component set is defined by a fault hypothesis (e.g. an unexercised statement may be faulty). A test suite is adequate with respect to some fault hypothesis if all of the component tests called for by the testing model have been produced. Test adequacy does not imply anything like ‘fitness for use’; nothing conclusive can be inferred about remaining defect density. Coverage provides a simple operational definition for an adequacy criterion. Statement coverage is the percentage of all executable source code statements in a program which have been exercised at least once by a test suite. Decision coverage (also ‘branch’, ‘all-edges’) requires that the outcome of each conditional statement be exercised at least once. Dataflow coverage is defined with respect to variable (object) definitions and uses; a define–use (DU) path is a path from the statement where a variable is defined to a statement where this definition is used.

Adequacy criteria have been ranked. A path test strategy $\alpha$ is said to subsume some other strategy $\beta$ if all the paths identified by $\beta$ are included in $\alpha$. The ‘all-DU-path’ set subsumes statement coverage (C1 coverage), decision coverage (C2 coverage), ‘all-uses’, and ‘all-definitions’. But since adequacy criteria are inconclusive as regards actual fault density, a coverage lower in this hierarchy does not indicate that a higher criterion is necessarily more fault-efficient and vice versa.

ADT coverage definitions have been developed and are discussed in Section 2 (Gannon et al., 1981; Choquet, 1986; Gaudel and Marre, 1988; Jalote, 1992; Zweben et al., 1992; Parrish et al., 1993a; Jia, 1993). Many of the surveyed sources call for ‘thorough’ testing without reference to any explicit coverage or adequacy criteria. In the absence of a commonly understood operational definition of ‘thorough’ testing (e.g. decision coverage), the only meaningful interpretation of a ‘thorough’ test suite is that it has satisfied someone’s curiosity.

Perry and Kaiser’s (1990) application of axiomatic adequacy criteria to testing subclasses establishes the scope of an adequate test suite with respect to inheritance but does not provide a specific test model (see Inheritance, above). However, this analysis has had a significant influence on subsequent work (McGregor and Sykes, 1992; Smith and Robson, 1992; Harrold et al., 1992; Murphy and Wong, 1992; Jacobson et al., 1992; Klimas, 1992; Berard, 1993; Turner and Robson, 1993a, d; Coleman et al., 1994; Graham, 1994; D’Souza and LeBlanc, 1994). As Perry and Kaiser’s result generally requires more testing, at least one technique has been developed to reduce the number of method-specific tests required for derived classes (Harrold et al., 1992).

3.7.1. Decision coverage

Several variants of decision coverage (Myers, 1979) and basis-path coverage (McCabe, 1976) have been used or advocated.

‘...test cases are created to execute each decision path in the code. ... except for paths that contained code for exception handling, test cases were written to ensure complete path coverage of each member function.’ (Fiedler, 1989).
Each 'decision-to-decision path' (a path between two decisions) should be exercised at least once; exercising pairs of decision paths may be useful (but time consuming) (Jacobson et al., 1992). Interrupts and exceptions must be considered in setting a coverage goal (Berard, 1993).

'... enough test cases must be written so that we can be reasonably assured that all statements are executed at least once, all binary decisions take on a true and false outcome at least once, all exceptions are raised at least once and all possible interrupts are forced to occur at least once.' (Berard, 1993, p. 259).

3.7.2. Optimistic scope

'Thorough' testing of base classes can reduce testing of derived classes (Cheatham and Mellinger, 1990). Classes are the proper subject of testing. Testing of base classes is similar to unit testing of modules in conventional languages. Unit testing may begin '... as soon as a class is implemented. If the class is not a derived class, the unit testing is equivalent to unit testing in a traditional system. ... If the class being tested is a derived class, the parent class should be thoroughly tested first. Then the derived class can be tested in conjunction with the base class.'

Message passing and inheritance influence how much testing will be needed. With unaltered inheritance, 'little additional testing [of a derived class] is needed. At most the interface should be retested.' An overridden or unique method '... must be tested as a new member function'. The meaning of 'thoroughly tested' is not discussed and the assertions about testing scope are not supported.

It is speculated that reuse of 'thoroughly' tested classes may reduce unit testing by 40 to 70%. System testing should also be eased since requirements are more readily mapped into implementation (compared to conventional implementations), but 'further study is needed.' The assertions made in this paper reflect an optimistic appraisal of the benefits of object-oriented development. Many of them are contradicted by later results.

3.7.3. Coverage for a classless language

Two adequacy criteria are posited for testing programs written in mXRL, a classless LISP-based object-oriented language (Trausan-Matu et al., 1991): component (similar to statement) coverage and message (similar to branch) coverage. '... in addition to the testing of procedural components, in OOP we have also to define procedures for testing the correctness of inheritance paths.' Adequate testing of a class requires (1) exercising all statements in each method; (2) all locally defined (not inherited) attributes 'must be accessed'; and (3) each feature obtained by a one-step inheritance link must be used at least once. It is asserted (without substantiation) that compared to conventional testing, 'Due to code sharing by inheritance, the amount of tests corresponding to the statement adequacy criterion may be significantly reduced.'

Branch coverage for mXRL is similar to covering 'all the different possibilities of code activation (i.e. branches)'. Message coverage requires (1) exercising all branches in methods; (2) sending all 'the possible different types of messages' that an object can accept; (3) sending a given message to all objects which accept that message; (4) exercising all inheritance links, including implicit (multi-level) inheritance links; (5) exercising all possible exits from methods composed by multiple inheritance.
3.7.4. ICpak test suite

The coverage goals set and met for testing of the Objective-C ICpak class library considered the effects of inheritance and dynamic binding. Each subclass was tested to demonstrate that "every method in all its superclasses was executable and gave correct results."

'With an object-oriented language one must verify that: all inherited methods used by a class are correct, all arguments that are subclasses of the specified argument type are correct, all methods by the same name perform the same logical operation, [and] the documentation is accurate and sufficient for an isolated user to use all the components.' (Love, 1992, p. 193).

3.7.5. Coverage checklist

Classes are not testable per se, but class testing is crucial for reuse so 'testing should be thorough ... completely unit testing a single instance, completely unit testing its corresponding class'. (Firesmith, 1992 and 1993b). Concrete, abstract, deferred and generic classes are each subject to unique errors. Generic classes (e.g. C++ template) 'may never be considered fully tested'.

'Changes to superclasses may have unexpected effects on the class to be tested. Changes to a superclass may therefore make test results obsolete and require significant regression testing, especially if configuration management and detailed analysis do not rule out impacts on the class to be tested. ... The primary purpose of classes and inheritance is reuse. For this reason, initial testing should be exhaustive and include stress testing.'

While the 'testing of objects is application-specific, the testing of classes should be more general because the developer of a class cannot know in advance how the instances of that class may be used on future projects.' (See Tables V–XII for the associated fault taxonomy). A complete object test suite requires the following.

(a) Every operation is executed.

Table V. Class/object faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not meet requirements</td>
</tr>
<tr>
<td>Abstraction violation</td>
</tr>
<tr>
<td>Timing problems</td>
</tr>
<tr>
<td>Incorrect state model</td>
</tr>
<tr>
<td>Invariant violation</td>
</tr>
<tr>
<td>Inheritance problems (class only)</td>
</tr>
<tr>
<td>Poor design/code</td>
</tr>
<tr>
<td>Concurrency failure</td>
</tr>
<tr>
<td>Incorrect create/init./destroy</td>
</tr>
<tr>
<td>Exceptions</td>
</tr>
<tr>
<td>Object not stored</td>
</tr>
<tr>
<td>Object stored in error</td>
</tr>
<tr>
<td>Doc./code inconsistent</td>
</tr>
<tr>
<td>Association missing</td>
</tr>
<tr>
<td>Violate design/code standards</td>
</tr>
<tr>
<td>Definition syntax error (object only)</td>
</tr>
</tbody>
</table>
Table VI. Message faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not meet requirements</td>
</tr>
<tr>
<td>Message sent to wrong supplier</td>
</tr>
<tr>
<td>Wrong message protocol</td>
</tr>
<tr>
<td>Wrong message priority</td>
</tr>
<tr>
<td>Message not in supplier</td>
</tr>
<tr>
<td>Parameter mismatch</td>
</tr>
<tr>
<td>Doc./code inconsistent</td>
</tr>
<tr>
<td>Failure to allocate requirements</td>
</tr>
<tr>
<td>Association not implemented</td>
</tr>
<tr>
<td>Violate design/code standards</td>
</tr>
<tr>
<td>Syntax error</td>
</tr>
</tbody>
</table>

Table VII. Exception faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unhandled exceptions</td>
</tr>
<tr>
<td>Exception missing</td>
</tr>
<tr>
<td>Exception incorrect</td>
</tr>
<tr>
<td>Exception not caught</td>
</tr>
<tr>
<td>Incorrect catch</td>
</tr>
<tr>
<td>Missing unit-level requirements</td>
</tr>
<tr>
<td>Supplier passes to client</td>
</tr>
<tr>
<td>Except propagating out of scope</td>
</tr>
<tr>
<td>Incorrect build</td>
</tr>
</tbody>
</table>

Table VIII. Attribute faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing Initialization</td>
</tr>
<tr>
<td>Out of range (type-check prevents most)</td>
</tr>
<tr>
<td>Unreachable states</td>
</tr>
<tr>
<td>Inappropriate transitions</td>
</tr>
<tr>
<td>Invariant violated</td>
</tr>
<tr>
<td>Pre/post violation</td>
</tr>
<tr>
<td>Values unnecessarily exported</td>
</tr>
<tr>
<td>Incorrect out</td>
</tr>
<tr>
<td>Failure to allocate requirements</td>
</tr>
<tr>
<td>Incorrect or missing unit</td>
</tr>
<tr>
<td>Incorrect accuracy</td>
</tr>
<tr>
<td>Incorrect visibility/scoping</td>
</tr>
<tr>
<td>Concurrent corruption</td>
</tr>
<tr>
<td>Syntax errors</td>
</tr>
<tr>
<td>Incorrect build</td>
</tr>
</tbody>
</table>

(b) All message parameters and exported attributes are checked using equivalence class samples and boundary values.
(c) Every out-going exception is raised and every in-coming exception handled.
(d) Every variable attribute is updated.
(e) Every state is achieved.
Table IX. Operation (method) faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message mismatch</td>
</tr>
<tr>
<td>Invariant violated</td>
</tr>
<tr>
<td>Pre/post violation</td>
</tr>
<tr>
<td>Incorrect operation performed</td>
</tr>
<tr>
<td>Unreachable code</td>
</tr>
<tr>
<td>Incorrect output</td>
</tr>
<tr>
<td>Exception not raised</td>
</tr>
<tr>
<td>Exception not handled</td>
</tr>
<tr>
<td>Incorrect state after except</td>
</tr>
<tr>
<td>Missed deadline</td>
</tr>
<tr>
<td>Syntax errors</td>
</tr>
<tr>
<td>Does not meet requirements</td>
</tr>
<tr>
<td>Incorrect priority</td>
</tr>
<tr>
<td>Incorrect serialization</td>
</tr>
<tr>
<td>Causes subsystem failure</td>
</tr>
<tr>
<td>Causes system failure</td>
</tr>
</tbody>
</table>

Table X. Inheritance faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class instantiated</td>
</tr>
<tr>
<td>Deferred resource not provided by</td>
</tr>
<tr>
<td>subclass of deferred superclass</td>
</tr>
<tr>
<td>Incorrect parameters in Generic C</td>
</tr>
<tr>
<td>Wrong feature inherited</td>
</tr>
<tr>
<td>Feature override missing</td>
</tr>
<tr>
<td>Feature delete missing</td>
</tr>
<tr>
<td>Incorrect dynamic classification</td>
</tr>
<tr>
<td>Incorrect hierarchy</td>
</tr>
<tr>
<td>Design standards violated</td>
</tr>
<tr>
<td>Does not meet requirements</td>
</tr>
<tr>
<td>Super class ripple-effects</td>
</tr>
<tr>
<td>Sub/super inconsistent</td>
</tr>
</tbody>
</table>

(f) Every operation is executed in each state (correctly where appropriate, prohibited where inappropriate).

(g) Every state transition is exercised to test assertions.

(h) Appropriate stress, performance and suspicion tests are performed.

Overall, '... prioritization should be based on the probability of finding bugs weighted by the severity of the resulting failure if the bug is not found.' Testing should proceed in step with the short-cycle incremental process. Increased reuse should be complemented by increased reliance on regression testing.

3.7.6. Incremental class testing

A minimal set of inherited C++ derived class features to be tested can be automatically determined (Harrold et al., 1992). Testing of base and derived classes is monitored. It is
Table XI. Scenario/subassembly faults (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not meet requirements</td>
</tr>
<tr>
<td>Incorrect export</td>
</tr>
<tr>
<td>Message/object mismatch</td>
</tr>
<tr>
<td>Message sent to destroyed object</td>
</tr>
<tr>
<td>Inconsistent garbage collection</td>
</tr>
<tr>
<td>Incorrect message/right object</td>
</tr>
<tr>
<td>Correct exception/wrong object</td>
</tr>
<tr>
<td>Wrong exception/right object</td>
</tr>
<tr>
<td>Constructor/destructor errors</td>
</tr>
<tr>
<td>Deadlock</td>
</tr>
<tr>
<td>Incorrect environment interface</td>
</tr>
<tr>
<td>Memory not reclaimed; inadequate memory due to leak</td>
</tr>
<tr>
<td>Concurrency problems</td>
</tr>
<tr>
<td>Friend function corrupts data</td>
</tr>
<tr>
<td>Design standards violated</td>
</tr>
<tr>
<td>Performance failures</td>
</tr>
</tbody>
</table>

*Note: A 'subassembly' is a group of objects that collectively service a single use-case, or use-case scenario.*

Table XII. Faults in aggregates (Firesmith, 1992)

<table>
<thead>
<tr>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect visibility</td>
</tr>
<tr>
<td>Missing component</td>
</tr>
<tr>
<td>Inconsistent component</td>
</tr>
<tr>
<td>Incorrect allocation/deallocation of resources</td>
</tr>
<tr>
<td>Does not meet requirements</td>
</tr>
</tbody>
</table>

*Note. An 'aggregate' is a group of objects which are used as a single entity and have a well-defined interface.*

argued that the selection criteria are consistent with the analysis of Perry and Kaiser (1990), but the aim is to reduce or reuse base class tests. A ‘thorough’ test suite is manually prepared and applied to each base class. Then each derived class is flattened. Required test cases for derived classes are determined by:

'... incrementally updating the history of the parent class to reflect the differences from the parent. Only new attributes or those inherited, affected attributes and their interactions are tested. The benefit of this technique is that it provides a saving both in the time to analyse the class to determine what must be tested and in the time to execute test cases.' (Harrold et al., 1992, pp. 78–79).

A new feature is ‘thoroughly’ tested at the first level it appears. If a derived class has a new feature, it is ‘thoroughly’ tested in the context of its defining class. Thus, lower level reuse of a feature will, in general, only need integration testing. ‘We first test base classes using traditional unit testing techniques to test individual member functions in the class.’ Procedure calls are stubbed-out or replaced with drivers. The specific test case design technique is not discussed, nor is an operational definition of a ‘thoroughly’ tested class provided.

The base’s test history (member function tested, test suite applied, test status) is saved.
When a newly derived class is ready to test, the base test suite is adapted to the subclass. The inherited features are not retested. Since each feature is tested independently of its role in the class, it is argued that the antidecomposition axiom is met.

A class uses graph is prepared to facilitate inter- and intra-class testing. 'For intra-class integration testing, we combine the attributes as indicated by the class graph and develop test cases that test their interfaces. ... we develop both specification-based and program based test suites.' The kind of specification used and the test case design technique are not discussed.

Inter-class interfaces are of two types: '(1) a member function in one class is passed an instance of another class as a parameter and then sends that instance a message; or (2) when an instance of one class is part of the representation of another class and then sends that instance as a message'. Inter-class testing follows the same approach used for intra-class testing. Integration test cases are marked apart from class test cases.

Several decision rules for setting the scope of derived class tests are presented: (1) for a new or untested feature in a derived CUT, a complete test suite is developed. Intra- and inter-class integration tests are prepared. A new data member is tested by '... testing [the CUT] with member functions with which it interacts'; (2) for inherited features, '... very limited retesting' is needed, since it is argued that the specification and implementation are the same. Integration testing is indicated by the anticomposition axiom if the CUT interacts '... with new or redefined variables, or accesses the same instances in the class's representation as other member functions'; (3) for redefined features, it may be possible to reuse the test cases prepared from specifications, but new program-based test cases will be needed. Conditions under which method-specific tests need not be repeated for inherited methods are presented. It is argued that this can significantly reduce the number of tests needed to verify inherited features, while still meeting Perry and Kaiser's (1990) adequacy axioms. This approach is referred to as 'hierarchic incremental testing' (HIT) in several subsequent reports.

3.7.7. Class Interface dataflow coverage

Dataflow coverage may be defined with respect to a synthesized method control flow graph (Parrish et al., 1993b). A graph model identifies 'define' and 'use' class methods, adapting the analysis for define-use ADT operations (Zweben et al., 1992). With the flowgraph, test sets can be identified that provide covers which correspond to various data flow path covers. These are listed in Table XIII. Since the flowgraph may contain

<table>
<thead>
<tr>
<th>Table XIII. Operation flow coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional code coverage</td>
</tr>
<tr>
<td>All statements</td>
</tr>
<tr>
<td>All branches</td>
</tr>
<tr>
<td>All paths</td>
</tr>
<tr>
<td>All uses</td>
</tr>
<tr>
<td>All definitions</td>
</tr>
<tr>
<td>All DU paths</td>
</tr>
</tbody>
</table>
infeasible edges owing to conditional flow, coverage may be unobtainable. A ‘weak class graph’ is defined where conditional edges are treated as a single edge (a ‘weak’ edge), assuring that at least one of the conditional edges can be covered. The approach of Zweben et al. (1992) is discussed in Section 2 and that of Parrish et al. (1993b) in Section 5.

3.7.8. Polymorphic bindings

Dynamic binding presents a coverage problem. Thuy (1992) argues that exercising a single binding of a polymorphic server is ‘insufficient: the coverage is complete only when all the redefinitions of the called method have also been exercised.’ However, this could require a large number of test cases which may be difficult to identify. Orthogonal arrays are suggested as a possible approach to selecting a subset of bindings to test by McGregor (1994a).

A component-directed strategy may be used to deal with the large number of tests needed to cover polymorphic servers (McCabe et al. 1994a). Each class in the system under test is tested separately by a driver. The class test suite must (1) provide decision path (cyclomatic) coverage; (2) exercise all intra-class uses of class methods; and (3) exercise each overloading of a polymorphic server at least once. When a class passes these tests, it is considered ‘safe’. It is argued that tests for a client of safe server classes need not retest polymorphic bindings encapsulated in the servers. It is also asserted that regression testing for clients of safe classes is not necessary in some circumstances. If the implementation of a safe class is changed (but not the interface or behaviour), no retesting of the client is needed. No empirical or analytical support is provided for these assertions.

3.7.9. State-based coverage hierarchy

A state-based approach to testing is presented by McGregor and Dyer (1993a) and McGregor (1994a). Details are discussed in Section 5. A hierarchy of state covers is presented. This adapts an analysis presented by Chow (1978) which proves certain properties of the ‘n-switch’ transition cover. Chow’s approach includes sequences to identify resultant states in the IUT. The ‘all-paths’ cover includes all possible sequences of some arbitrary length; this is typically infeasible. The ‘switch’ cover exercises all paths that begin and end in the same state. (Path length is limited to edges comprising a circuit. Arbitrarily long cycles are not included). A transition cover exercises each transition at least once. A state cover visits each state at least once. An event cover accepts each event at least once. Figure 7 presents McGregor’s subsumption hierarchy for these relationships.

3.8. Fault taxonomies

3.8.1. Debugging fault taxonomy

The use and misuse of typical features of object-oriented programming languages can contribute to ‘bugs’ (errors, faults and failures) (Purchase and Winder, 1991). A bug taxonomy is provided along with a discussion of features for debugging tools. Inheritance and dynamic binding present many opportunities for errors. Table XIV summarizes this
taxonomy. The categories are defined in terms of *history* (the generic fault: where in the software lifecycle the bug was introduced), *deviation* (the generic failure: observable incorrect behaviour or state) and *mindset* (the generic human error causing the fault).

### 3.8.2. Component/fault taxonomy I

A fault taxonomy guides a white-box testing strategy for mXRL (Trausan-Matu *et al.*, 1991). mXRL is implemented in Common Lisp. It is a classless language which implements inheritance via delegation and slots. A slot is similar to an instance variable, but is inherently polymorphic—any kind of object can be bound to a slot. The test strategy is based on a taxonomy of errors that can result due to this kind of late binding.

- **Inheritance loops.** For example, with classes A, B and C, A hasSuper C, B hasSuper A, C hasSuper B.
- **Reference to non-existent slot.**
- **Duplicated slots.** Two or more sibling classes define a slot of the same name and value.
- **Redundant slots.** The same slot appears in several subclasses without being defined in a common superclass.
- **A subclass incorrectly redefines a superclass slot.**
- **Wrong value in slot.**
- **Message sent to object without corresponding method.**
- **Missing object** (referred to, but not defined).
- **Unused object** (defined, but no reference).
- **Incorrect design.** Same slots (semantics) in different objects not related by inheritance.
- **Incorrect method under dynamic binding.**
- **Incorrect method under multiple inheritance,** due to synonyms, naming error, or misuse of name resolution.
### Table XIV. Bug taxonomy of Purchase and Winder (1991)

<table>
<thead>
<tr>
<th>Bug type (fault class)</th>
<th>History (fault)</th>
<th>Deviation (failure)</th>
<th>Mindset (error)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptual Specification</td>
<td>Initial requirements definition</td>
<td>Application problem not solved</td>
<td>Inadequate problem analysis</td>
</tr>
<tr>
<td></td>
<td>Design</td>
<td>Requirements or specification not met</td>
<td>Communication failure</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Design of top-level objects and protocol</td>
<td>Poor class structure, unorthogonal protocols, Inheritance hierarchy sprawling, inconsistent</td>
<td>Inadequate knowledge of OO design techniques, Premature hierarchic factoring or levelling</td>
</tr>
<tr>
<td>Algorithmic</td>
<td>Design of top-level objects and protocol</td>
<td>Incorrect method output</td>
<td>Inadequate knowledge, Specification misunderstood</td>
</tr>
<tr>
<td>Reuse</td>
<td>Design</td>
<td>Unexpected, undesired behaviour in reused components</td>
<td>Implementation incorrect</td>
</tr>
<tr>
<td>Logical</td>
<td>Implementation</td>
<td>Incorrect method output</td>
<td>Misuse of reuse component, ad hoc extensions of inheritance</td>
</tr>
<tr>
<td>Semantic</td>
<td>Implementation</td>
<td>Abend due to type mismatch with late binding</td>
<td>Inadequate programming knowledge or misuse of target environment services</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Implementation</td>
<td>Incorrect output, abend</td>
<td>Inadequate programming knowledge, typographic error</td>
</tr>
<tr>
<td>Domain adherence</td>
<td>Runtime</td>
<td>Runtime exceptions, message search failures</td>
<td>Inadequate knowledge of application or target services</td>
</tr>
</tbody>
</table>

### 3.8.3. Component/fault taxonomy II

A proposal to generate test cases by an unspecified automatic source code ‘search’ is based on the observation that objects preserve state and typically accept any sequence of messages (Smith and Robson, 1992). Four generic errors may lead to a corrupt state. The four errors and the search strategies constitute a fault taxonomy.

1. **Interroutine conceptual**: incorrect design resulting in overlapping function between methods. No elaboration is offered as the technique is strictly code-based.
2. **Interroutine actual**: state may be corrupted under certain message sequence patterns. The proposed test strategies suggest three tests.
   1. **Identity sequences**: Sequences of methods which should result in no net
change to an object may be incorrect. This strategy is explored by Doong and Frankl (1994).

(ii) **Set-get sequences.** Test cases are to be produced in which state changing and reporting methods are paired.

(iii) **Random interleaving.** An ‘exhaustive’ testing strategy is patterned after the work of Jalote and Caballero (1988) to exercise ‘... all legal combinations of routines supported by an object, to a sufficiently great depth of combinations.’

(3) **Intraroutine conceptual:** incorrect design resulting in missing methods. No elaboration is offered as the technique is strictly code-based.

(4) **Intraroutine actual:** state may be corrupted due to incorrect capability in a method and incorrect output or abend.

(a) Memory leaks are likely. For languages that do not provide automatic garbage collection, tests to ‘ensure that an object frees all memory used in its lifetime’ should be generated.

(b) Inheritance anomalies are limited to subclass specializations (this position is contrary to many other views). Classes are flattened, and subclass methods that are identical with the parent are excused from testing.

3.8.4. **Component/fault taxonomy III**

Generic faults in objects, classes, messages, attributes, methods, inheritance, scenarios and clusters each have an effective general test strategy (Firesmith, 1992). The fault categories for each component are listed in Tables V–XII. For each fault type, either inspection, white-box testing, black-box testing, integration testing, or regression testing is indicated as the appropriate verification technique.

3.8.5. **Component/fault taxonomy IV**

A prototype of a static analyser for object-oriented code, VOOPS (verification of object-oriented programming systems) classified faults by abstraction, encapsulation, modularity and hierarchy (Hayes, 1994). This list is presented in Table XV.

4. **AUTOMATED MODEL VALIDATION**

4.1. **Overview**

Automated model validation seeks to improve an abstract model by transforming it into an executable but typically incomplete implementation. The model may be a formal or semi-formal representation. The implementation may be used as an exploratory prototype, may participate in a simulation or may be exercised by a test suite. It is used to refine, verify and validate requirements or specifications which are subsequently transformed into a suitable programming language.

Execution of a representation often provides useful insight into the dynamics of a system which are otherwise hard to imagine. This facilitates early and less costly removal of omissions, inconsistencies, incompleteness and infeasibility.

Automated model validation is not unique to object-oriented development. Many
Table XV. VOOPS fault taxonomy (Hayes, 1994)

| Abstraction               | Class contains non-local method |
|                          | Incomplete (non-exhaustive) specialization |
| Encapsulation            | Public interface to class not via class methods |
|                          | Implicit class-to-class communication |
|                          | Access module data structure from outside |
|                          | Overuse of friend/protected mechanisms |
| Modularity               | Method not used |
|                          | Public method not used by object users |
|                          | Instance not used |
|                          | Excessively large numbers of methods |
|                          | Too many instance variables |
|                          | Excessively long method |
|                          | Module, class, method, instance variable not used |
|                          | Module contains no classes |
|                          | Method contains no code |
|                          | Class contains no methods |
|                          | Class contains no instance variables |
|                          | Excessively large instance variable |
|                          | Excessively large module |
| Hierarchy                | Incorrect branching |
|                          | Dead-end or cycle |
|                          | Incorrect multiple inheritance |
|                          | Improper placement |

Approaches have been studied and implemented (Agresti, 1986). Prototyping is typically used to facilitate requirements elicitation. Operational specification uses an executable specification language. After validating system behaviour, some or all of the implementation may be produced from an operational specification.

'An executable specification language is a specialized programming language. Each executable specification language is intended to serve as the foundation for a streamlined software development process in which only one complete formal description of the system is written. If any other representation is needed, it is generated automatically from the original description. Thus the executable specification may replace several descriptions (such as requirements specifications, design specifications, and implementation code) that are written independently by hand . . . It should be obvious that no language will ever be able to play all these roles for all systems.' (Zave, 1991, p. 212).

In a transformational implementation, problem models and specifications are processed by program generators to produce the entire implementation. In contrast, operational specifications are generally independent of a specific problem domain. The transformational approach relies on a specific problem domain model to generate an implementation.

'The transformations embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the transformation process is executable code that is guaranteed to be consistent with the given problem specification (Smith and Para, 1993, p. 60).
Automated model validation is not yet widespread in object-oriented development. This section reports on automated model validation for formal models (ADTs) and two semi-formal object-oriented models. While the first four reports are not purely object-oriented, they are arguably object-based. The semi-formal models include a simulator for an Ada implementation and two approaches to object-oriented requirements modelling.

The systems validating formal models (ObjEx, Ina Jo, SITE and LOFT) serve two purposes. They can validate system models and, with few or no changes, they can automatically produce test cases for implementations. In contrast, additional effort is needed to apply the validation suites produced for semi-formal models to subsequent implementations.

4.2. Formal models

4.2.1. Design time testing in OBJ

OBJ is an algebraic specification language for ADTs (Gerrard et al., 1990). The basic elements of the language are objects and theories. Objects define constructions; theories state requirements. An OBJ object can be executed by the ObjEx tool. The approach is motivated by the difficulty and expense of formal proof for requirements. Testing provides a less costly means for validation.

An OBJ theory is a general, implementation-independent statement of behaviour. Theories define behaviour using set functions. An OBJ construction (object specification) is derived from a theory. It is an algebraic ADT containing type definitions, variable definitions, exported operations and axioms expressed as equations whose terms are operations on the type.

The test strategy is to: (1) develop constructions—a construction is produced for each set function in the theory. These functions give invariants for the ADT (its 'requirements'). The requirement object is translated into an executable implementation of the invariant. It also functions as a test oracle since it is paired with a construction object to check the results of the construction object; (2) devise test cases for the construction object; and (3) execute the requirement object in conjunction with the construction object. If the result produced by the construction object is consistent with the requirement object, the test passes.

For example, a program is to be developed that will produce an indented list of a Unix file directory structure. The construction object FLATTEN produces the list. The requirements for FLATTEN include the set function IS-TREE-LIST. This takes an indented list as an argument and returns a boolean value that indicates whether a given indented list is actually a list. The test code is of the form:

\[(\text{for all directories}) \text{ IS-TREE-LIST?}(\text{FLATTEN(directory)})\]

The test cases consist of various patterns of directory structures selected by the user. For a given directory structure, the test passes if the indented list meets the criteria for a list. There is one test case for each such constraint in the requirements.

Test case selection depends on the regularity and uniformity assumptions (Weyuker and Ostrand, 1980; Bougé et al., 1986). First, the regularity hypothesis assumes that require-
ment constraints will hold for all values of the type if they can be shown to hold for all values of ‘complexity’ \(k\) or less, where \(k\) is the number of primitive operations on the type. Since there are five operations on the type, a five level directory structure in the example meets the regularity hypothesis. The uniformity hypothesis assumes that all domains are uniform, i.e. there are no subsets in the domain which will result in abstractly different behaviour, which implies that picking any arbitrary set of domain inputs will not result in a significant omission. The logic programming approach to test data generation uses the same assumptions.

### 4.2.2. Ina Jo

Two systems are presented and compared for automatically validating Ina Jo specifications (Kemmerer, 1985). Ina Jo is a model-based formal specification which uses data types. Automatic validation is performed by generating testable prototypes and by symbolic execution. This reduces the cycle time for discovery of an error or omission in a specification. Without early verification, errors can be propagated to the implementation, requiring rework of the specification. Ina Jo is a:

‘... non procedural assertion language that is an extension of first order predicate calculus. The language assumes that the system is modeled as a state machine. The key elements of the language are types, constants, variables, definitions, initial conditions, criteria and transforms. A criterion is a conjunction of assertions that specify the critical requirements of a good state. A criterion is often referred to as a state invariant since it must hold in all states including the initial state.’ (Kemmerer, 1985, p. 34).

An Ina Jo model has two parts. Functional requirements consist of a start predicate \(F_{\text{start}}\), a sequence of operations and a result predicate, \(F_{\text{result}}\). The start predicate may be ‘true’, allowing any operation in any state, or may be much more restrictive. The predicate and operation expressions may constrain variables. The result predicate defines the set of states obtained from the operations. This triple specifies a resultant state \((F_{\text{result}})\) and the states from which the transformation may be entered \((F_{\text{start}})\).

Specification elements include definitions for types, constants, variables, state invariants, and initial conditions. Transforms are non-procedural first-order predicate calculus definitions of state changing operations. Operations have signatures.

Two properties are of interest for a model. A specification for a functional requirement is satisfiable if some implementation can produce the desired functionality. A specification for a functional requirement is valid if every possible implementation can produce the desired functionality. A valid specification means that any implementation will be correct, but this is typically ‘too cumbersome and the verification too costly when dealing with real systems.’ (Kemmerer, 1985, p. 34) A satisfiable specification means that the implementation can be shown to meet the functional requirements by executing them. Two contrasting tools are presented for the assessment of these properties.

The first generates a testable prototype in a procedural language (Pascal) from the non-procedural Ina Jo representation. It generates and executes testable implementations. It generates test cases, but the user must provide domain boundaries.

‘... since the resulting program represents only one of many possible consistent implementations, a successful test shows only satisfiability. In the same manner, an unsuccessful test shows that the functional requirement tested is not valid, but does not indicate if it is satisfiable or not.’ (Kemmerer, 1985, p. 39).
The symbolic execution tool does not compile the specification. Symbolic execution is accomplished by substituting expressions without assigning specific values to variables. States are described by predicates instead of by enumeration. The tool accepts a user specified starting state. It checks the initial predicate and then executes transforms specified by the user. The resultant state expression is compared to the final state predicate. The consistency of the expressions can be automatically evaluated. An inconsistent symbolic expression reveals an error. Additionally, it can ‘... check the resultant symbolic values to see if they define the desired set of resultant states.’ (Kemmerer, 1985, p. 33).

A comparison of the relative advantages and disadvantages of these two tools is given in Table XVI (a minus indicates a disadvantage, a plus, an advantage).

4.2.3. The SITE system

The 'specification and implementation testing' (SITE) system is an automated approach to checking the completeness of abstract data type specifications (Jalote and Caballero, 1988; Jalote, 1989 and 1992). The purpose of the system is to detect missing axioms in an ADT specification. A missing axiom may be due to coding error or an omission in the problem model represented by the specification.

The approach uses only syntactic specification elements (the variable definitions and operation signatures, excluding the axioms). It is based on the observation that a complete

<table>
<thead>
<tr>
<th>Table XVI. Testable prototype versus symbolic execution (Kemmerer, 1985)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testable prototype</strong></td>
</tr>
<tr>
<td>Translation of infinite domain</td>
</tr>
<tr>
<td>Translation of non-determinism of transformation</td>
</tr>
<tr>
<td>Satisfiability property</td>
</tr>
<tr>
<td>Number cases checked per input</td>
</tr>
<tr>
<td>Debug</td>
</tr>
<tr>
<td>Validation</td>
</tr>
<tr>
<td>Interpretability</td>
</tr>
<tr>
<td>Tractability</td>
</tr>
</tbody>
</table>
and consistent set of axioms should not allow the production of an instance (by any sequence of state-changing operations) which cannot be acted upon by an accessor operation. If such a state obtains, the specification has been shown to have an omission or inconsistency.\(^1\)

The strategy is to generate, by simple permutation, sequences of modifiers terminated by an accessor. The number of operations in each sequence is limited to a practically small value. It is argued that test sets composed of short modifier sequences (2 or 3 operations are mentioned) are both computationally feasible and effective for revealing errors. Use of the technique in a classroom setting has demonstrated effectiveness in revealing specification and implementation errors (Jalote, 1992).

The test case generation algorithm can be summarized as: (1) generate, to a user-selected ‘depth’ all sequences of modifier operations. Depth refers to the number of ADT expressions to be nested in a test case. A depth of two is argued to be practical since most ADT axioms are not more deeply nested so doubly nested test cases are sufficient to detect their errors; (2) permute arguments: substitute external names for inline expansion; and (3) apply each accessor to each member of the set of modifier sequences. Thus, 32 test cases are required if there are two accessor operations and four modifier operations.

Nested operations are evaluated by replacement. For example, the nested expression:

\[
\text{empty (append(new(), new()))}
\]

creates two null queues, concatenates them, and then reports whether the result is empty or not.

This formulation is used to check the completeness of an algebraic specification composed of operations and axioms. The axioms represent constraints on the results of operations. For example,

\[
\text{empty(new()) = true}
\]

The technique tests for incomplete axioms by ‘... showing that there is an instance of the data type for which the specifications do not define the value for some [accessor] operation.’ This is done by generating modifier operation sequences followed by an accessor, e.g.

\[
\text{empty( new()).}\quad \text{front( delete (add(q1)))}
\]

The approach seeks to establish ‘sufficient completeness’ of the axioms within practical bounds. This requires that all accessor operations are defined for all possible instances of the type which can be produced by any sequence of modifier operations. Computing whether or not a set of axioms is \textit{consistent} and \textit{complete} is undecidable. A more modest goal is set: to detect ‘incompleteness caused by missing axioms.’ (Jalote, 1989, p. 528).

\(^1\) Jalote calls operations which map type values onto other types ‘behaviour operations’ (e.g. from a stack to an integer or boolean). Operations that produce or change a type instance are termed ‘non-behaviour’ (e.g. concatenating two strings to form a third). The corresponding object-oriented terms are used here: \textit{accessor} for behaviour operations, \textit{modifier} for non-behaviour.
This simplification means axiom relationships need not be used to generate test sets. This approach incurs some limitations.

\[\ldots\] it is very important to consider the semantic part as well, since different instantiations of arguments in a sequence, corresponding to different paths through an ADT tree, can lead to profoundly different abstract states of the specification. Thus, it is necessary to select many different paths through the ADT tree arising from a given original sequence, or equivalently, to choose values of parameters that exhibit different relationships to one another. (Doong and Frankl, 1994, p. 126).

The user selects initial values for the ADT under test. Results are evaluated by inspection. No details are provided for either activity.

4.2.4. LOFT system

The LOFT system produces black-box test cases for ADTs (Dauchy et al., 1993)—see Section 3. It was used to test two software controllers for subway train doors. In this application, the platform side doors may be opened under slow speed before halting. Once opened, no acceleration is allowed and the train must stop within a certain distance of opening (it must not pass the end of the platform). The emergency brake is to be activated if any of these limits are passed.

LOFT was ‘used to analyze the specification during its development. The tool was used to compare and test new versions of the specification module against the previous ones.’ (Dauchy et al., 1993, p. 243). The generated tests suggested some additional ‘integration testing scenarios’ which the subway manufacturer had not planned to test. The details of LOFT are discussed in Section 2.

4.3. Semi-formal models

4.3.1. Ada cruise control simulator

Simulation can deal with some problems in testing an automotive cruise control written in Ada (Jones, 1990). The system implements a dynamic control strategy where continuous response in time and frequency is a necessary capability. Simple testing of point values cannot reveal problems with under- or over-damping, dynamic instability or improper gain calibration.

An initial implementation exhibited poor control response: oscillations and undesirable settling time. Validation of the control strategy by simulation on a prototype would have revealed these problems early in development. Ada encapsulation is well-suited to simulation since it can provide stable type interfaces while allowing implementation changes. Simulation of the plant (a vehicle) proved to be more difficult and complex than the implementation of the controller itself. The simulator was enhanced with tracing and reporting capabilities. Three lines of simulation code were written for each line of the cruise control code.

Several recommendations are made: (1) software engineers should not focus exclusively on software. Understanding the dynamics of the controller and the plant proved to be essential in developing a correct controller. The initial understanding of input–output mappings and discrete state control was insufficient to accomplish this; (2) test early, not
late. Early development of a simulator will provide the greatest cost leverage. If the specification is represented as an Ada program, an early implementation prototype can be produced concurrently with the development of the simulator; and (3) learn how to model the problem, as well as the solution. In the cruise control application, modelling the problem required an understanding of both control system theory and vehicle dynamics. Jones (1990) observes that the amount of time required to develop the test software does not depend on how well you write the delivered software, or on the size of the system under test.

4.3.2. Shlaer/Mellor OOA

The Shlaer/Mellor (S/M) object-oriented analysis methodology partitions a requirements model into several domains: application, architecture and implementation. This scheme is essential to the approach, facilitating concurrent development and validation by the separation of concerns (Mellor and Shlaer, 1994a, b).

Domains are orthogonal, consistent abstractions with well-defined interfaces at each boundary. This facilitates domain-specific automatic validation. If this separation of concerns is absent, high and low levels of abstraction are easily mixed, increasing the scope and difficulty of integration during modelling and implementation. However, as domain subjects must be mutually exclusive, it is argued that integration of domain implementations is facilitated. Separation of concerns by domains also means that development and validation of the models may be done concurrently. There is a validation strategy for each domain.

The application domain is checked for correct syntax. Behaviour test cases are manually identified and run to 'see what happens.' A test case is a:

'... scenario, thread of control, or use case, defined by an input, a specific system state, a new system state and possibly a set of outputs. An input that starts off a test case is an unsolicited external event, which is defined as an event generated by some external entity that was not caused by some previous action of the system.' (Mellor and Shlaer, 1994a, p. 12).

Values for state variables and determinant variables (transition guards) are identified by thread path analysis. This can be manually simulated, but automation is much more effective. The model can be executed on several commercially available CASE tools. Automation can be simple enactment, or provide tracing, breakpoint, stepping and state inspection. Pass/fail evaluation is subjective.

'If the system does not behave in the desired manner, as verified by our own understanding as well as the clients [sic], then we must modify the analysis models. Then we move on to another unsolicited external event and repeat the entire process until we have completely covered the problem space.' (Mellor and Shlaer, 1994a, p. 12).

The architecture domain is application and implementation independent. It is an abstract virtual machine that provides three main services: event handling, data access and combinations. Each data access feature is to be tested. Data access is supported by iterator methods; some suggested tests include: finding a single instance matching when one, none, or several are present, finding several instances (returning a list) under the same conditions, and trying client usage of these operations. Combinations are class-to-class interfaces which control component activation. An active object has a state machine for
each instance. Assuming the state controller works, tests are to be developed for events between active objects, events to self, and for accepted and rejected events. Assigner objects have a single state machine for all instances, typically to manage contention. Tests are suggested for events between active objects, events to self, data access messages to passive objects, data access messages to active objects and data access messages to other assigner objects. Accepted and rejected events are to be tested.

Architecture model enactment uses the same case tool simulation, driven by manually prepared test cases. This architecture model is ‘very simple minded and blindingly uniform’ (Mellor and Shlaer, 1994b, p. 22). This has an advantage. ‘As the complexity or variation in the architecture increases, so does the cost of testing and integrating it. The worst possible case, however, would be where each class is handcrafted, one by one.’ (Mellor and Shlaer, 1994b, p. 22).

The implementation domain uses an unspecified translation mechanism to produce an implementation from the application and architecture domains. The translation mechanism can be automated: ‘... verification of the translator then is the same as the verification of a compiler.’ This may be done independently of application and architecture validation. No tools are discussed for this transformation.

It is suggested that model validation test cases be reused for integration testing of the actual implementation. Several extreme assertions are made about the degree of confidence these techniques offer. A verified model, mechanisms and state specification mean that ‘we know that all events generated and received in the implementation must be correct.’ (Mellor and Shlaer, 1994b, p. 20). If you ‘verify the behavior of the application and of these iterators, and it is possible to guarantee that the iterator is constructed according to the standard logic, then we know that all data accesses in the implementation must be correct.’ (Mellor and Shlaer, 1994b, p. 20). No argumentation or evidence is offered in support of these claims.

4.3.3. Real-time object-oriented modelling (ROOM)

Real-time object-oriented modelling is an object-oriented methodology explicitly designed to be executable (Selic et al., 1994). It provides modelling constructs for embedded real-time systems and fully supports the object-oriented paradigm.

The concept of a scenario is fundamental. A scenario is a particular pattern of externally determined events to which the system must respond. Typically, these events constitute a task meaningful to a user. For example, a user places a telephone call from a PBX. The PBX needs to treat each digit key press as an event, recognize when a valid number has been entered, etc. However, all this is subsumed by the ‘place a call’ scenario. Scenarios can also be constructed to model exceptions, resource degradations and partial system failures. Concurrent scenarios can be modelled and simulated.

Actors are the source of events to which the system must respond. For example, the PBX user is an actor. Actors are the portal for scenarios. Actors are explicitly modelled in ROOM.

An executable ROOM model may be created from an incomplete ROOM representation. Figure 8 shows the process which produces the executable model. The hybrid model is the model as it exists in the CASE repository. This is translated to the linear form model, where all attributes and relationships are expressed as strings, in a kind of p-code. This may contain some expressions specific to the target language. The Language L model
results from translating the linear form model into a particular source code (e.g. C++). This is compiled with additional source files as needed. The link step binds ROOM enactment objects and target environment run-time libraries to produce the executable model.

Developers may interact with the model in a number of ways. They can input individual test cases or can specify simulated actors (sources of input) which automatically produce an input stream. Validation actors can be added as subclasses to increase controllability and observability of message threads. The validation strategy is iterative and incremental, and does not insist on a nearly correct or complete model.
'Model construction and model validation alternate during development. ... Model validation is strongly associated with model construction. ... validation through operational models occurs as early as possible. A modeler does not have to analyze all the requirements or construct all of the model before validation can occur.' (Selic et al., 1994, p. 424).

The main steps in model validation are: (1) identify scenarios; (2) specify validation actors; (3) execute the model to debug message structure; and (4) execute the model to verify component behaviour. It is suggested that this process be applied to packages of scenarios, until all the system requirements have been completely modelled and validated. No specific criteria for completeness or coverage are defined.

Validation actors are added to the model to drive and trace scenarios. They provide repeatable tests and can be organized per concurrent scenario. Interface simulation allows actors to send and receive messages through an interface, so validation actors can be exchanged or inserted in an event stream. These actors can monitor or corrupt message flow, or simulate various hardware interfaces. Validation actors can be embedded in the system at any point in the message structure. This increases observability and controllability without disrupting system structure.

The structural framework of a requirements model (class-to-class message interfaces) determines whether a thread completes. Debugging this structure is usually the first order of business. The quickest approach for the modeller is to use validation actors to drive scenarios and trace resulting message threads to identify the point of failure.

After the structure is verified, behaviour debugging may begin. The states of key actors may be viewed simultaneously; important state diagrams may be animated during execution. Halts and failures can be simulated by turning off an individual actor or a group of actors. The modeller can designate breakpoints for individual states or transitions. This allows error logging when an event is sent to a component in a state in which the event cannot be accepted.

'The basis of validation is the execution of scenarios, driven by validation actors. Such actors are usually created on a per-scenario basis, so that scenarios can be run either individually or concurrently to uncover scenario interaction problems. Specific validation components outside or inside the system can be used.

Validation takes advantage of interface simulation to stub out actors or to emulate various component failures. Inheritance allows variants of models with validation components to be cleanly separated from pure design models. Multiple containment allows validation components to be simultaneously part of a validation system and the system under examination.' (Selic et al., 1994, p. 422).

5. TEST CASE DESIGN

5.1. Overview

Test case design produces test cases using a test model based on a fault hypothesis. A test case specifies test state, test input, expected output and expected state. A test model provides an algorithm or heuristics to create test cases from a specification or implementation.

Strategies for test case design are implementation-based (also structural or white-box), representation-based (also functional or black-box), or hybrid (grey-box). An approach
may be *formal*, if its fault-revealing capability has been established by formal (mathematical) analysis, or *heuristic*, if it is guided by expert judgement or past experience. Black-box testing usually implies a specification-based approach but is also used to describe tests derived from analysis of implementation interfaces.

Testing strategies can also be classified by the scope of the implementation under test. A *unit test* exercises a relatively small, independently executable software component. A unit in object-oriented software is typically a cluster of classes, with one class being the unit under test. An *integration test* exercises some related collection of units. In object-oriented software, this collection is typically a functional subsystem composed of several clusters and reused classes from a library of basic classes. A *system test* involves all components comprising an application system and typically includes all or most of the target environment's virtual machine.

The surveyed sources are presented within these categories, chronologically by date of publication.

### 5.2. Implementation-based class testing

The smallest standalone unit for testing is an instantiation of a single class. Since class instance variables are typically instances of other classes, testing a single class typically requires some minimal integration. The class under test and this minimal set of servers is referred to as a *class cluster*. Class testing usually refers to class cluster testing.

Three approaches follow a similar ‘state-based’ approach for C++ programs (Turner and Robson, 1993d; Hoffman and Strooper, 1993b; Kung et al., 1993). Data member usage is analysed to identify value sets which are modelled as states. Each uses a different approach to identify transitions. Test sequences are derived by adapting the transition tree cover (Chow, 1978). In the work of Parrish *et al.* (1993b), a path model similar to a state machine is synthesized by considering operations exported by the class under test. The balance of the approaches employ various heuristic and path-based strategies.

#### 5.2.1. C++ unit test (A)

A unit test strategy for C++ programs used in a life-critical health care application adapted the basis path technique (McCabe, 1976) to identify test cases for member functions, followed by domain analysis to assign specific values (Fiedler, 1989). The goal was to ‘ensure complete path coverage of each member function.’ Classes derived from ‘well covered’ base classes received less testing than those with an uncovered base. Tests of the ‘signals or exceptions that are raised (not propagated) by each function’ were also devised. Test cases were also identified by tracing member function interfaces: inheritance, using (call) hierarchy, constructor/destructors and exception handlers.

‘... defects have been discovered by applying associativity rules to member functions. That is, if string s1 is null, and string s2 is not null, s1 > s2 should yield the same results as s2 < s1. In addition, the use of the object itself as a member function input parameter proved valuable in uncovering subtle implementation errors. For instance, given s1 is a string, the test s1.Append(s1) [is useful]’
5.2.2. mXRL

A test strategy for programs written in mXRL, a classless, LISP-based language is presented by Trausan-Matu et al. (1991). The test strategy is simply to access and update all instance variables (slots) in each object. Since slots are typically references to other objects, this test strategy also exercises dynamic binding. The content of these tests is discussed above under Section 3.8: ‘Fault taxonomies’.

5.2.3. FOOT proposal

The ‘framework for object-oriented testing’ (FOOT) is a proposal for generating test cases from source code (Smith and Robson, 1992). The approach is limited to detecting faults resulting from inconsistent programming. No sort of conformance-directed testing is considered. Several ‘search’ strategies are suggested to extract test cases: exhaustive, inheritance, memory, identity and set/examine.

Exhaustive test case generation is based on Jalote and Caballero’s (1988) ADT strategy (see Section 2). A test case consists of a sequence of member functions to be executed. For example, the ADT sequence empty( new()) is restated as C++ member functions:

\[
\text{Queue(int 10), IsEmpty, ~Queue}
\]

This is claimed to be ‘exhaustive’ testing: ‘... all legal combinations of routines supported by an object, to a sufficiently great depth of combinations. ... Truly exhaustive testing can be achieved if the class has built-in constraints ...’ (Smith and Robson, 1992, p. 48). This is an unfortunate choice of words. Except for trivial cases, exhaustive testing is impossible. No definition or discussion of ‘sufficiently great’, ‘truly’, or ‘constraints’ is provided. The implications of the absence of C++ constructs equivalent to ADT axioms are not discussed.

Inheritance tests are generated from flattened classes. Subclass methods identical with the superclass are excused from testing. ‘Regression analysis’ is suggested as a means to further select methods, but it is not clear what this means. Memory tests are to be generated for languages that do not provide automatic garbage collection to ‘ensure that an object frees all memory used in its lifetime.’ Identity tests are to be generated based on a ‘search’ to identify sequences of methods that should result in no net change to an object. The mechanics of the search are not described. Set/examine tests are to be generated so that methods which change and report states are exercised in pairs.

The identity and set/examine test suites would appear to be subsumed by the so-called exhaustive test suite. No specific algorithms are offered for preparing test cases, nor are any coverage criteria defined or referenced. The tool would not allow ‘tester guided’ tests of any sort. Some speculation is made about the applicability of data flow testing, but ‘further research will be required.’

5.2.4. Test strategy for reuse

A testing strategy is outlined as part of a process that aims to promote reuse (McGregor and Sykes, 1992). The discussion begins by asserting that Weyuker’s axioms as interpreted by Perry and Kaiser (1990) are not applicable to testing object-oriented software. The authors sketch the arguments that later appear in the work of Harrold et al. (1992) and elsewhere as the ‘HIT’ strategy for reducing method-specific testing under inheritance.
Some general guidance is offered for test case content and organization. It is suggested that 'all major states' of a class should be exercised. No specifics are provided for test case design. The test plan for a class should:

'(1) Exercise each method in the class, (2) Use values chosen using the preconditions of each method, (3) Consider interactions of the methods, and (4) Account for all states of the class.'

(McGregor and Sykes, 1992, p. 218).

5.2.5. C++ unit test (B)

A comprehensive test process was used for a commercial CAD/CAM system composed of over 1 MLOC of C++ (Thuy, 1992). To test derived classes, both inherited and locally defined methods (member functions) must be exercised, but:

'... systematically testing all methods is then generally wasteful. On the other hand, to test only the defined or redefined methods of a class is not always sufficient. ... one must test the methods defined or redefined by the class, plus all the inherited methods that use directly or indirectly a method redefined by the class.'

Public and protected methods are tested for specification conformance. Private methods are white-box tested. ‘Structural’ coverage (not defined) is attained for all methods. Abstract classes require special treatment since concrete methods must be provided. This may be complicated if a deferred method is used by a concrete method.

‘In order to simplify the test management, it is preferable to test all these methods in the same unit test, using a derived concrete class redefines none of them, if such a one exists. If not, one should find a minimal set of concrete derived classes that cover all these methods.’

Two cases are identified for template classes: (1) the template class does not use methods in the class of the type parameter. Only one template instantiation need be tested; and (2) the template class uses methods in the class of the type parameter. All possible template instantiations must be tested as individual classes, except when the generic arguments of an instantiation are derived from those of another instantiation and redefine none of the used methods. Other aspects of this approach are discussed in sections on fault hypothesis, integration, testability and test process.

5.2.6. Testgraphs

An automated state-based approach to testing uses a state model of the CUT (the testgraph) which is represented in a driver class (Hoffman and Strooper, 1993a, b). A corresponding oracle class is programmed. The oracle determines test pass/fail by comparing the state of the CUT to the expected state in the testgraph class. A driver class obtains test sequences from the testgraph class and uses the oracle class to check the results. The test sequence is applied to both the CUT and the testgraph class, providing a kind of parallel execution.

This research does not focus on test case design methods or how they may be automated, which is ‘far more difficult’ than achieving repeatability by automation.

'The driver class is based on a testgraph which partially models the CUT as a state machine, but with vastly fewer states and transitions. The oracle class provides essentially the same operations as the CUT, but supports only the testgraph states and transitions. Surprisingly
thorough testing is achievable with simple testgraphs and oracles. The key is designing the two together, to avoid tests for which input generation and output checking are unaffordable.’ (Hoffman and Strooper, 1993b, p. 472).

The testgraph is composed of states (nodes) and transitions (arcs). Examples of testgraphs are provided, but no general modelling techniques are mentioned. A trace (a sequence of member function calls) defines each transition and state. A transition arc trace uses only modifiers to compute the destination state. The state node trace uses only accessors to report state. A testgraph can be ‘parametrized’ allowing a ‘family’ of tests, i.e. any number of trace sets can be applied to the same structure.

‘While the CUT state space is normally very large, the testgraph state space is vastly smaller. Roughly speaking, the testgraph state space contains all states that can be reached; the testgraph state space contains those states that will be reached by the test suite.’ (Hoffman and Strooper, 1993a, p. 86).

The oracle is incomplete. ‘... to build a general oracle that does a complete check on the elements is hard ... To reduce the cost of building and maintaining the oracle, we choose a ‘partial’ oracle that does convincing cross checking, but at substantially lower cost than a full oracle. ... we expect that it will catch most incorrect implementations.’ (Hoffman and Strooper, 1993a, p. 89).

Hoffman and Strooper (1993a) demonstrate the approach on the authors’ integer set class. Hoffman and Strooper (1993b) report the results of testing the integer set class from a commercial class library. The tests revealed a significant fault.

5.2.7. C++ data scenarios

The states of C++ data members may be analysed to develop an implementation-based test suite (Turner and Robson, 1993a, b, c, d). The approach is motivated by the following observation:

‘When a class is defined, in the vast majority of cases, the only explicit (or implicit) order in which the features can be used 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 calls. This possibly infinite order is tested by ... exercising each feature with all possible starting states.’ (Turner and Robson, 1993a, pp. 12-13).

A substate is the value of a particular data member at a point in time; a state is the value of all data members at a point in time; a partial state is a set of substates over some of the data members. Substates are identified by analysis of data member usage. For example, \(x = 0\) implies two substates: 0 and not 0. The analysis is similar to that employed for equivalence class partitioning. In general, there will be as many states as there are combinations of substates (some combinations may be ‘don’t care’ situations).

A data scenario is a ‘situation which [is] significant to the model upon which the data structure is based.’ For example, the data scenarios for a list are: empty; a list with a single node; current pointer at first node; and pointer to last node. Data scenarios can be used to refine the substate models.

Testing ‘combinations of substates’ (states) may be problematic. Each data member
may have a large number of states and associated test cases. However, tests for aggregate states are ‘more difficult to generate.’

Transitions model the effect of member function activation. No technique for identifying illegal transition sequences is discussed, as ‘classes should be written with no implied order for the calling of features.’ (Turner and Robson, 1993d, p. 307).

Expected data values are to be derived from a specification. The class under test is to be instrumented so that the state of each data member can be reported. ‘At least one new feature per substate is required. These will enable the tester to inspect the value of the chosen substates.’ For example, in C++, a ‘printf’ statement at entry and exit can list member function arguments. Assertions are recommended. The test process has twelve steps.

Step 1. Posit one substate per data member.
Step 2. Determine class data scenarios.
Step 3. Define additional substates implied by the data scenarios.
Step 4. Design substate specific and general values.
Step 5. Add trace member functions to the CUT.
Step 6. Add monitor member functions to the CUT.
Step 7. Define expected input and output from the CUT specification.
Step 8. Identify the CUT’s inter-class calls.
Step 9. Devise a bottom-up test plan for the CUT.
Step 10. Identify ‘from the design . . . each state that the [member function] is expected to handle [and] the state that the [member function] should leave the object in after the call.’

Step 11. Code the results of Steps 1–10 in a test script file. Use the MKTC tool to exercise the CUT and record the results.
Step 12. Repeat from Step 10 until all member functions have been tested.

This technique is to be augmented with additional ‘structural and functional’ testing. When used with an LCSAJ (Linear Code Sequence And Jump) path coverage analyser, less than ‘100% sub-path coverage’ was obtained.

5.2.8. Test cases by automatic reverse engineering

A white-box, state-based testing strategy for C++ uses a block branch diagram (BBD) for each member function (Kung et al., 1993 and 1994). This representation is a decorated flowgraph used for both ‘structural’ and ‘functional’ test case design. It is used to identify paths for branch coverage. When a predicate node contains multiple conditions, each condition is enumerated to allow testing of each condition combination. ‘Functional’ test cases are derived by identifying boundary values and typical interior points for input parameters and manually computing the expected results.

An object state diagram (OSD) is derived (automatically) from the source code. Diagrams for base and component classes are done first. A finite state machine model for each data member of the CUT is synthesized.

‘A state of a data item is a range of values which is derived from the decision vertices that evaluate the value of the data item and lead to updating of the data item. For example, if \( x > 5 \) is a decision vertex, then, from this two ranges of data, i.e., two states, for \( x \) can be derived [state 1 ranges from \(-\infty \) to 4; state 2 ranges from 5 to \(+\infty \)].’ (Kung et al., 1993, p. 209).
A symbolic program is generated to identify functions that change individual data members. These symbolic functions are represented as state transitions. The individual data member state transition models are then merged to yield an OSD for the class. No details of this process are provided. Test cases are generated from the state model by applying Chow’s algorithm (Chow, 1978).

5.2.9. Intra-class flowgraph

An interface dataflow test suite can be developed from ‘a general conceptual framework for conducting flow graph modeling of classes’ (Parrish et al., 1993b). This approach is an adaptation of two ADT approaches (Zweben et al., 1992; Parrish et al., 1993a)—see Section 2—to object-oriented programming. The approach is strictly implementation-based. A model and a process are presented to compensate for the absence of the kind of formal specification used by Zweben et al. (1992).

‘An important result of this framework is the insight that conventional flow graph-based techniques can be applied to classes as either program or specification-based techniques. This means the techniques may be applied even in the absence of a formal specification, thus making them applicable to current-day development practice.’ (Parrish et al., 1993b, p. 96).

Exported (externally visible) class methods are represented as nodes in a flowgraph. Edges represent allowable sequences of method activation. The absence of an edge represents an infeasible sequence. Without a specification to the contrary, it is assumed that all sequences are allowable; i.e. excluding constructors and destructors, any method may follow any other method, including itself. Since this approach eschews specification information, a completely connected graph will result for all classes.

Node and branch coverage of this graph are shown to be analogous to node and branch coverage of a conventional flowgraph. Dataflow paths (between method arguments) can be inferred from the class interface, if it has (1) all method names, (2) all method arguments, (3) type and transmission (‘in’, ‘out’, ‘in–out’). Each ‘in’ or ‘in–out’ argument is modelled as a definition; each ‘out’ or ‘in–out’ argument is modelled as a use. Path sets analogous to conventional ‘all-use’, ‘all-definitions’, and ‘all-DU paths’ (excluding cyclic paths) criteria can be derived.

A weak class graph is suggested as a strategy to deal with difficulties posed by conditional paths. Complete coverage of all edges is not required. Instead, if at least one of a set of conditional edges is traversed, the weak coverage criterion is met. While allowing that this is limited testing ‘... the weak criteria at least tests certain interesting combinations of operations and has the virtue of being fully automatable.’

This approach is limited in that only method parameters are considered; such a test suite could cover interface dataflow and miss intra-class DU flows among instance variables (Harrold and Rothermel, 1994).

5.2.10. Modifier sequences

A modifier method is one which may change the value of its object. Changes in the state of an object are the result of sequences of modifier methods. Eschewing a guided approach to selecting method sequences, Parrish et al. (1994) simply enumerate sequences
of modifier methods, so that 'as many relevant combinations of operations are examined as possible.' The approach is applicable to classes with increasingly 'complex' states, that is, collections which may contain an arbitrarily large number of members.

A tree of sequences is generated, limited to some arbitrary depth. The root node represents the constructor. For each lower node, there are edges to nodes which each represent one of the modifier methods in the CUT. For example, the tree for a class with insert and delete modifiers is given in Figure 9.

Test suites are generated by walking the tree and sending the indicated messages. Test data values are randomly generated. Two sequence selection strategies are discussed. A breadth-first test sequence chooses modifiers left to right, by level. For example, the breadth-first sequence to the third level is:

- NI, ND
- NII, NID, NDI, NDD
- NIII, NIID, NIDI, NIDD, NDII, NDID, NDDI, NDDD

Instead of beginning each sequence from new, an object in the desired state is copied, and the next (single) operation is applied. The 'thread' test sequence uses a modified depth-first selection strategy which is not explained in detail. States (the result of a sequence) are equivalent if an equality operator reports them as such, so NIID = NI. It is noted that eliminating equivalent states would cut down the number of tests, but it is too expensive to find all equivalent states.

Student programs for four data types were tested by the tool. This showed that 'A breadth-first approach to state generation may show defects more quickly than generating states in threads for each individual operation. However, the threads approach may be more useful in isolating the location of a particular defect.'

5.2.11. Safe classes

The 'safe' approach provides a strategy to deal with the potentially large number of tests needed to cover bindings of polymorphic server classes (McCabe et al., 1994a, b). Each class in the system under test is tested separately by a driver. For a class under test to be considered 'safe', a driver must execute a test suite that meets three goals: (1) decision coverage is obtained for each method in the CUT; (2) all server object methods used by the CUT are called at least once and each possible binding for a polymorphic server object is used at least once; (3) if the CUT exports polymorphic

![Figure 9. Modifier method tree](image-url)
methods, the driver must exercise, at least once, each possible binding of each method. A class is considered ‘safe’ when it meets all three criteria and all of its server classes are also safe. This dictates a bottom-up (uses) order of integration.

It is argued that clients of safe servers need not re-exercise bindings of the server’s servers. This limits the scope of unit-level integration for a class under test to its immediate servers, avoiding a potentially very large set of bindings for integration.

It is also asserted that regression testing for clients of safe classes is not necessary in some circumstances. ‘A change to the implementation, not the interface, of an existing method within a system requires only that the changed method be tested. . . . When a SAFE class is extended through derivation and inheritance, it only needs to be tested in the weakest sense.’ (McCabe et al., 1994b, p. 25). Since the interface is the same and client usage of this interface has already been tested, client uses do not need to be re-tested. A driver is used to run a test suite on the new implementation. This means it is not necessary to re-test each client usage of the interface in an application system. It is argued that this is a practical necessity when widely used server classes are changed.

5.2.12. Testing for incorrect pointer aliases

Incorrect pointer aliases are difficult to reveal by flow or state-based testing (D’Souza and LeBlanc, 1994). An alias occurs when ‘. . . two or more pointers . . . point to the same object when they should not be doing so.’ (D’Souza and LeBlanc, 1994, p. 37). This typically is a result of mis-naming. Several examples of subtle alias faults are provided. It is argued that integration testing to reveal alias faults is difficult and expensive. An approach is presented to assist in early detection (during class unit test) of alias faults.

The test strategy has two main steps. First, the class under test is instantiated and exercised by a subclass test driver in conjunction with a method activation trace (no specific test case design approach is suggested).

Then, a second program searches the test execution traces for possible aliases and presents them to the tester. The trace generated during testing captures each object reference made by the OUT. For each reference, the trace contains the call path to the target object, the target object id, and its ‘dynamic type’ (i.e. the class to which it was bound at the point when the associated path reached it). The trace file is sorted on object.id. This groups all occurrences where two or more data members point to the same object. These groups are printed out. Then ‘. . . the tester can decide whether the sharing, if present, is intended and appropriate.’

A lengthy polemic characterizes state-based and ADT sequence testing as ‘inefficient’ and ‘indirect’ because these techniques might not reveal alias faults.

5.2.13. Generic classes

There are two forms of generic class: a parametrized class which takes a data type (a class) as a parameter to create a new type-specific class (a template class in C++) and an abstract superclass that defines an interface with no implementation. Generic classes cannot be tested per se. One must generate a concrete instance and apply a test suite to that instance (Overbeck, 1994b).

The testing model in the work of Overbeck (1994a) is used to show that generic classes must be instantiated to be tested and to show the limits on reduction of testing for subsequent instantiations. The primary results are: (1) the scope of unit testing for an
instantiation of an abstract class is the same as it is for any non-generic class; (2) the scope of integration testing for an instantiation of an abstract class is the same as it is for any non-generic class; (3) a parametrized class must be instantiated to be unit tested. Assuming these tests adequately exercise features defined by the class, it is not necessary to repeat these unit tests on every subsequent instantiation of a parametrized class; (4) however, each new instantiation requires some testing, since it is necessary to test all uses between the parametrized class and the class of the type parameter for each instantiation of the parametrized class; (5) the client uses of each instantiation of a parametrized class must be tested. A client using a type \( P \) instance which has passed tests of this usage does not obviate the need to test uses of a type \( Q \) instance; (6) it is not necessary to repeat unit and integration tests of a generic class when a parametrized instantiation and its servers are (re)used in a new context.

5.2.14. Intra-class dataflow testing

Dataflow paths over an entire C++ class may be identified with the use of a ‘frame’ (Harrold and Rothermel, 1994). While dataflow determination is straightforward for an individual member function, intra-class, inter-function dataflow requires a model of how member functions interact. With such a model, it is possible to identify how data member definitions in one function may be used in other functions. In effect, the model represents a generic client of the class under test.

The technique is entirely code-based, allowing testing in the absence of a specification. This is argued to be a good thing, in that specifications may not be available and specification-based testing is not guaranteed to reveal all faults. The test model has ‘the further advantage of guiding testers in the selection of sequences of member functions that should be run, and sequences of member functions that need not be run.’ (Harrold and Rothermel, 1994, p. 158).

Three kinds of dataflow are defined: (1) \textit{intra-method}, where a define–use pair occurs within the scope of a member function; (2) \textit{inter-method}, where a define–use pair occurs over two member functions, or over successive calls to the same member function issued as a result of a call from a client to a public member function in the server; and (3) \textit{intra-class}, where a define–use pair occurs over two public member functions, or over successive calls to the same public member function, i.e. when a sequence of public member functions is used.

A graph model is used to identify these flows and to construct a driver class called a \textit{frame}. The calling structure of the CUT is placed in the frame. The frame calls each public member function of the class under test. The graph is established by substituting successively more detailed graphs for higher level nodes. The member function nodes in the class graph are replaced by class flowgraphs. An entry and exit node for each member function is added to the call site in each member function flowgraph. ‘A frame is an abstraction of a main program \( P \) in which calls to public member functions are selected randomly by a switch statement \( S \), where \( S \) is enclosed in a loop \( L \).’ (Harrold and Rothermel, 1994, p. 160).

The frame allows def–use pairs to be computed by the PLR algorithm (Pande et al., 1994). ‘To compute intra-class def–use pairs, we must consider the interactions that occur when sequences of public member functions are invoked.’

Aliasing (different variables pointing to the same object) causes problems in tracing dataflow, and is seen as warranting further study.
5.3. Representation-based class testing

5.3.1. Gauges

Cox offers a unique point of view on testing object-oriented programs (Cox, 1988 and 1990; Cox and Novobilski, 1991). Testing is seen as an activity which establishes conformance to a class specification by use of a ‘gauge’ which is a testable ‘specification language’. No details are provided for any aspect of this language.

It is often difficult to get a correct and unambiguous understanding of what a software component does (Cox, 1988). A combination specification/test capability could help. ‘The specification language compiler . . . doesn’t generate code, but tests whether code written in some implementation language meets its specification by invoking test procedures from a library.’ Integrating classes that have been debugged individually is problematic. A test harness is useful for preliminary integration. ‘The automatically inherited TestTools [a driver for the class under test] often detect longstanding bugs that have persisted because nobody thought to verify . . . deeply inherited methods, particularly easily forgotten object-level methods . . . overridden to adapt to the peculiarities of the subclasses.’

Mapping the specification, the object under test, and its implementation is problematic. ‘Inheritance is only occasionally useful as a specification tool, but is far more often used as a tool for automating implementation. This leaves the specification inextricably mingled with the implementation of the class and its superclasses.’ If the implementation inheritance changes, the same specification/test tool will probably not retest new inherited components. This approach does not address method level or larger aggregation testing issues.

‘The formal testing model . . . was almost right; that is, right in the sense of being a useful tool for practical problems. The right part was that the TestTools developers built what amounts to a database of individual procedures for testing some aspect of a class’ behaviour in a procedural notation that could be applied directly to the implementation under test. The wrong parts were (a) the determination of which tests should be applied to which classes was solely based on an implementation matter; and (b) it left the specification implicit in the code, rather than capturing it explicitly.’ (Cox, 1988, p. 47).

Since implementation need not mirror functionality, Cox (1990) argues that ‘just as a measuring stick is not a higher level saw, a specification tool is not a higher level implementation tool.’ They should be treated differently, according to the roles they play. A specification language should include test procedures. A test procedure takes a single argument that designates the implementation to be tested. The test procedure runs the implementation to see if it complies with an associated specification, or ‘gauge’. This would be implemented by a specification language compiler (Cox and Novobilski, 1991), which would:

‘. . . preserve specifications by preserving the gauges that determine compliance to the specification. Just as implementation language compilers generate implementations from libraries of subcomponents, the specification language ‘compiler’ assembles pre-existing test procedures to form executable gauges that determine whether a putative implementation complies to its specification.’ (Cox and Novobilski, 1991, p. 97).

The Objective-C ‘assert’ macro offers a ‘rudimentary’ implementation of a specification/test language (Cox and Novobilski, 1991). Assertions can be used for ‘white-
box' testing, and grouped into pre- and post-conditions. Pre-conditions 'verify that the conditions the method depends on were met by the caller. For example, verify that argument values and/or types are as they should be for the method to work properly.' Post-conditions 'verify that execution achieves the expected result.' The test procedures discussed earlier are argued to be 'black-box' since assertions may be placed in a separate file. This allows independent development of assertions.

5.3.2. ASTOOT

ASTOOT (A Set of Tools for Object-Oriented Testing) is a research system used to test programs written in Eiffel (Frankl, 1989; Doong and Frankl, 1991, 1992, 1994). The conceptual basis for the approach is presented by Frankl (1989). Effective reuse requires highly reliable components. Automated testing leads to higher reliability by allowing more testing. Automation is needed since manual inspection of the state of complex types is slow and difficult. ASTOOT is based on the idea that:

'... the natural units to test are classes, and that in testing classes, one should focus on the question of whether a sequence of messages puts an object of the class under test into the 'correct' state.' (Doong and Frankl, 1994, p. 102).

Operations are designed to be interleaved; hence state must be correct under any sequence. Testing for the production of the correct states must consider operation sequences.

'... it is possible to test a class by testing each of its methods individually, treating each as a function mapping some input space to some output space... However, so doing shifts the focus of testing away from the essence of data abstraction—the interaction among operations.' (Doong and Frankl, 1994, p. 106).

This approach adapts ADT testing strategies to object-oriented software (see Section 2). The ASTOOT system automatically generates paired sequences of method operations which should yield the same observable results. Each sequence in the pair is executed. The paired objects are compared. If they are not equivalent, the test has succeeded in revealing a fault. For example, Table XVII shows how test cases 1 and 2 run a pair of sequences on stack objects A and B, which should yield the same result in all four sequences—5 on top.

The evaluate-and-compare strategy obviates the need for an oracle to produce expected results for test cases, solving a generally difficult problem. However, the CUT must

<table>
<thead>
<tr>
<th>Test</th>
<th>Test sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Stack A</td>
<td>create, push(5), push(6), pop</td>
</tr>
<tr>
<td>Stack B</td>
<td>create, push(5)</td>
</tr>
<tr>
<td>2 Stack A</td>
<td>create, push(5), push(6), pop, top</td>
</tr>
<tr>
<td>Stack B</td>
<td>create, push(5), top</td>
</tr>
</tbody>
</table>

Table XVII. Equivalent sequence test cases (Doong and Frankl, 1994)
implement a method that can compare two class instances and determine if they are equivalent. This built-in checking may be problematic. If functionally equivalent classes have different implementations, it may be difficult to determine equivalence.

A base-class specification and test suite can be used to reduce derived class testing. With the base class tested, it is necessary to test the derived class against the derived-class specification and the base-class specification. The derived class is expected to respond to inherited messages as does the base class, but with its own data state. A composite ADT expression (realized as nested method invocations) can obviate several problems with test drivers. A standalone driver must initialize the object state, run a test sequence, then report the resulting state. By using expression sequences, initialization and reporting the application code is performed within the sequence.

Testing begins by entering a manually prepared formal ADT specification for the class under test (CUT) in the Lobas language. This is checked for correct syntax. A 'driver generator' automatically produces test driver code from the ADT specification. The driver is an instance of a driver class which activates the CUT using the current test case.

The ASTOOT system was used to test two classes written in Eiffel (stack and queue) with known faults. Several thousand test sequences were automatically generated, executed and evaluated in a few minutes. Experiments were performed to investigate the effect of variation in sequences: the number of operations from 10 to 100, ranges of data values over three orders of magnitude, and used four proportions of add and delete operations. Overall, longer sequences and wider data ranges were reported more likely to reveal faults. On average, longer and wider sequences were able to reveal the fault about 50% of the time. Shorter, narrower sequences were roughly half as effective.

5.3.3. Black-box strategies

Equivalence class partitioning and boundary value analysis should be used for ‘white-box testing’ of methods (Siegel, 1992). ‘Complete and thorough testing’ is advocated for each method, intra- and inter-class using relationships and new or modified derived methods. No specific techniques are offered for identification of these test cases, nor are any coverage metrics defined or referenced. The inherent flexibility of object-oriented programming languages can enable complex functionality. Several ‘black-box’ strategies are outlined to verify functionality.

1. **Usage profiling.** A component usage frequency profile is developed to set testing priorities.

2. **Objective architecture mapping.** Consideration of the physical packaging of the system may suggest test cases. This is similar to error-guessing (Myers, 1979).

3. **Random statistical shotgun.** It is suggested that test cases be developed by ‘random sampling’ of using relationships. Fault-prone relationships are to be sampled more frequently. Sampling techniques are not discussed.

4. **Directed mutation analysis.** Test cases are developed by permuting the domain, sequence, or frequency of some baseline test case. The permutation is to be done ‘randomly’, but permutations corresponding to high usage frequencies are preferred. No specific randomization technique is suggested. This is a scheme for test value selection, not program source code mutation.
5.3.4. **OOSE testing strategy**

The object-oriented software engineering (OOSE) approach covers development from initial concept definition through system installation (Jacobson et al., 1992). Testing is an integral part of OOSE. The test strategy follows generally accepted practice and describes techniques for unit, integration, system and regression testing. Early test planning is advocated (the V model).

A battery of white-box and black-box test case design techniques are to be used and organized around the *use case*. The use case is a key concept in OOSE. ‘When a user uses the system, she or he will perform a behaviourally related sequence of transactions in a dialogue with the system. We call such a special sequence a use case.’ (Jacobson et al., 1992, p. 129).

Unit testing begins with method path analysis. Decision coverage (Myers, 1979) is recommended. However, unit testing is:

‘... a test of a larger unit than in a traditionally developed system. Integration testing, however, is carried out at an early state, since communication is essential for the system development. All objects have predefined interfaces which also contribute to less dramatic integration testing.’ (Jacobson et al., 1992, p. 82).

Inheritance, polymorphism, dynamic binding and overriding can make testing difficult. Inheritance can result in several contexts for a method, requiring a test for each context. OOSE is in agreement with the work of Perry and Kaiser (1990). ‘In the worst case, we may need to develop unique test cases for every level in the inheritance hierarchy.’ Polymorphism results in more paths to test. An overloaded message should be treated as a case statement, for purposes of path analysis.

Equivalence class partitioning (Myers, 1979) is used to identify input and output data. When variables are instances of classes, ‘object flattening’ is suggested to develop equivalence classes. Points from the equivalence classes of used objects are substituted for the object reference (flattened) to prepare the test plan. This allows explicit consideration of the collective domain of an object.

5.3.5. **Partitioned GUI state model**

Finite state models of GUI behaviour are intractable, so a testing approach that considers states and their transitions (input events) one at a time is needed (Loughman, 1992). The approach is reported to have been used ‘with a pure object-oriented language’, but no details are given. An *object-response chart* is used to define event/response sequences. For each response (display), an *attribute pattern chart* is used to show acceptable combinations of object (display widget) state. A table enumerates valid combinations of widget states, providing a kind of testing checklist.

5.3.6. **State-based functional testing**

A state-based approach to class testing is presented by McGregor and Dyer (1993a, b), McGregor (1994a, b) and McGregor and Korson (1994). (These sources differ primarily in the degree of detail used in examples; the test strategy is consistent). The goals of this test approach are (1) to make best use of a scarce resource, 'the human oracle that must
certify the correctness of each test case, or in the worst case, each test result; (2) to produce the minimal number of test cases needed for a given coverage; (3) to use an adequacy criterion that allows incremental increases in coverage; and (4) to use existing workproducts as the basis for test cases.

A state model of a class is developed from an OOA/D specification or by examination of the CUT. This model is a ‘blend’ of several approaches for state modelling. In particular, the focus is on design state which is ‘a set of vectors of observable values that share some behavioral attribute of interest.’ States are disjoint; substates must be disjoint partitionings of single super-states. Methods are modelled as events. A transition is defined by method input parameters and object state, a method, and method output parameters and the resultant object state.

The approach requires that the CUT conform to strict inheritance. This constraint allows reuse of state-based test cases by lower levels of a class hierarchy. It is met when: (1) pre-conditions for subclass methods are the same or weaker than their superclass pre-conditions; (2) post-conditions for subclass methods are the same or stronger than their superclass post-conditions; and (3) every method complies with the class invariant and all superclass invariants. Several patterns of strict inheritance and their implications are discussed by McGregor and Dyer (1993a). The concept of strict inheritance is generally similar to the formally defined concept of monotonic subtypes (Choi and Mannino, 1991; Leavens, 1991).

A graphical representation similar to statecharts (Harel, 1988) is used. The statechart concurrency notation is adapted to provide a graphical shorthand for an ‘automata product’. The model is not a ‘formal finite automata’ since the two or more transitions for any state may accept the same method (event) and result in different states; however, a notation is provided to represent conditional transitions (McGregor and Dyer, 1993a). This is dropped in later versions and the same event is allowed on two or more transitions from a state.

With the state model in hand, the test strategy proceeds in several distinct steps: (1) test each constructor; (2) test each accessor; (3) for each method, design and run test cases to check each post-condition for each method, including tests to force post-condition failure and exception handling; (4) perform transition testing, beginning with each constructor and traversing every transition in the state representation. All test cases should check the class invariant.

Balancing of coverage goals and testing cost is discussed. The ‘HIT’ technique for reduction of intra-class test cases under inheritance (Harrold et al., 1992) is suggested (McGregor and Korson, 1994). The number of bindings between clients and polymorphic servers can be large; orthogonal arrays are suggested as a means of selecting bindings to test (McGregor, 1994a). However, no details are provided for path modelling, selection or sensitization.

5.3.7. FREE approach

The ‘flattened regular expression’ (FREE) approach to testing object-oriented systems models classes and clusters as a state machine, identifies test sequences from this model and selects test values to cover the state space of the CUT (Binder, 1994b). Flattening means that all inherited features are considered, providing a view necessary for adequate testing of inherited features (Perry and Kaiser, 1990). This is representation-based responsi-
bility testing, since the state model is derived from the class specification. Test sequences are generated to cover all state transition cycles for the class under test. These sequences are regular expressions of the flattened class state machine, hence the name.

The FREE state model is a Mealy type machine with extensions for conditional transitions and may be mapped into a statechart hierarchy. Inbound messages and encapsulated interrupts are modelled as events. Outbound messages to servers and returned messages to clients are modelled as actions. Class state-space is defined by points constructed from minima and maxima of class attributes. The state-space model may be used to select test points or to assess state-space coverage. A technique for selecting test points from the state-space produces a test set that grows in direct proportion to the number of class attributes, avoiding a combinatorial explosion.

Implementation-based class testing is supported by the FREE-flowgraph, which represents intra-class, inter-method flows. This graph is synthesized by combining method flowgraphs and the FREE state model. This means, for example, that dataflow paths among all class methods can be identified and instrumented for coverage.

A modal class has constraints on allowable message sequences; these constraints are represented by the FREE state model. Test suites for modal classes are generated by walking a tree produced from the class state model (Chow, 1978). Each test case (a set of inputs applied to a known state with an expected result) is a leaf of this tree. With this tree structure in place, adding more test cases (leaves) is straightforward. Leaves may be added to meet any desired testing goal: branch coverage, domain boundary coverage, exception coverage, etc. The fault-finding power of individual test cases is leveraged by embedding them in a sequence which will reveal control faults.

A non-modal class has either trivial or no constraints on message sequence. An interface data-flow model is developed for non-modal classes. Message sequences are selected from this model to provide set/get pair coverage. Iterator methods are exercised at least twice.

5.3.8. Test cases from OMT models

Test cases can be automatically produced from a specification in a CASE repository (Poston, 1994). An OMT (object modelling technique) representation was used (Rumbaugh et al., 1991). However, additional effort during analysis and design are required to make ‘test-ready’:

‘A model is test-ready when a tester can create from it occurrences or instances of data items, events and states which cause logical conditions to be true and false and actions to be performed.’ (Poston, 1994, p. 50).

The OMT meta-model required extension as well. ‘The definitions in OMT object models are not complete enough for testing [because they] do not require data domain definitions.’ (Poston, 1994, p. 50). Similarly, a test-ready OMT dynamic model (state model) has a definition for every state, action and transition pre-condition. The CASE tool facilitated production of the extended models and captured information necessary for automatic test generation. The boundary value data is entered into the CASE tool. The OMT model must observe naming conventions that equate to a C++ implementation.

An OMT specification augmented with boundary value information for objects and attributes was exported in IEEE 1175 format. This file contained class names, interface boundary value specifications and the uses structure of the CUT. It was processed by a
test case generation tool \((T)\) to produce a file of test cases. Every OMT action and every external event had one test case. Test data values were generated by boundary value analysis using a proprietary algorithm to select value combinations.

This process was applied to an example system with one GUI window and five classes. Minimal specification coverage required 347 test cases. The total time to prepare the automated test suite was estimated to have been at least twenty times less than manual production of the same tests.

5.3.9. Method sequence specification

A method sequence specification \((\text{MSS})\) is a regular expression whose symbols represent class methods (Kirani and Tsai, 1994). It represents sequential method activation constraints, which allow automatic consistency and completeness checking of classes related through inheritance. \([\text{It}]\) can be used to generate test cases from the method sequence specification.' (Kirani and Tsai, 1994, p. 28).

An MSS represents 'the sequence in which the methods can be executed, such as method \(m_1\) before method \(m_2\).' A formal notation is presented for MSS. The authors propose that MSS be embedded in each class along with an MSS checker. An algorithm is presented which determines whether or not any message received complies with the sequential constraints of the MSS. If MSS are consistently embedded in a class hierarchy, then correct usage of superclass methods in a subclass can be automatically determined:

'The child class sequence specification is consistent with all the parent class sequence specifications if any valid method sequence in the parent class is a proper subsequence of at least one method sequence of the child class.' (Kirani and Tsai, 1994, p. 34).

The set of methods used by each client of a class is not necessarily the same. The subset used by each client may be represented in the MSS. Then, the MSS checker can also check that a message from a particular client belongs to those allowed for that client. It is asserted that an MSS can be used to generate test cases, but no details are given.

5.4. Integration testing

This section presents several general integration test strategies. Several integration test reports are provided in the experience reports section (Jüttner et al., 1994a, b; Arnold and Fuson, 1994; Murphy et al., 1994).

Integration of object-oriented systems is a complex task (Jüttner et al., 1994c). There are at least three levels of integration which typically proceed in parallel: (1) class; (2) hierarchy/library; and (3) application system and regression testing. Library integration should consider future uses in addition to specified capabilities. The focus of testing is mainly method interaction and object communication. Inheritance, call/use relations, and data aggregation cause problems. 'In general, it is not possible to do the integration testing according to only one of these structures without a lot of tedious stubbing. Therefore one needs a flexible mixed strategy.' (Jüttner et al., 1994c, p. 13).

Harrold et al. (1992) assert that intra-class techniques for test suite reduction under inheritance can be applied to inter-class testing but provide no discussion. The regression test strategy discussed by Rothermel and Harrold (1994) and reviewed in Section 5.6
bears on integration testing. McCabe’s Safe strategy (see Section 5.2.11) results in uses-ordered integration.

5.4.1. Basic integration concerns

‘Big-bang integration will fail miserably in an OOD environment.’ (Siegel, 1992). Instead, small chunks should be programmed, unit tested and integrated in a shorter cycle. The focus of integration testing is defined by six main issues:

‘... do we have the right classes, are any classes missing, are the classes at the right level, have we included the latest versions, do the classes have the correct visibility one to another, and do the methods work?’

5.4.2. OOSE integration strategy

A coherent approach to integration testing is used in the OOSE methodology (Jacobson et al., 1992). Integration occurs several times during development and should (ideally) be performed in the target environment. Integration test cases are designed to verify that collections of implementation objects can cooperate. It is suggested that collections of objects follow requirement or specification aggregations: use cases, service packages (configuration units) or blocks (class clusters).

‘When a block has been tested, you test it together with another block. When these two are working together, you add another, and so on. When all appear to work, we can consequently change to black-box testing ...’ (Jacobson et al., 1992, pp. 324-325).

Integration testing is organized by use cases because they ‘explicitly interconnect several blocks and classes.’ (Jacobson et al., 1992, p. 327). Their specification provides the basic elements of a state machine: stimulus, response and next state. This is used as a checklist to see if all pairs of transitions are correctly implemented.

‘... integrate one use case at a time ... The requirements model forms again a powerful tool here; as we test each use case, we check that the objects communicate correctly [and check] user interfaces ... the requirements model is verified by the testing process.’ (Jacobson et al., 1992, p. 327).

In addition to normal usage patterns, ‘odd courses’ for use cases are to be tested as well as use cases identified from requirements and user documentation. Integration by use cases is also suggested by Firesmith (1992) and Graham (1994).

5.4.3. Architecture-level integration

A hierarchic, incremental integration strategy is appropriate and needed for very large systems (Thuy, 1992). With a very large system (over 1 MLOC of C++) integration testing should begin as soon as possible, and must deal with incompleteness. The alternative, waiting for all components to be in hand (big-bang integration), is problematic for several reasons. The entire system must be available and running, which typically happens late in development. Debugging at this stage is difficult and time consuming.

A hierarchic approach to design and testing is necessary. At the lowest level is a
**package** of classes (any cohesive logical or physical group). A package is a kind of high level module. Member functions which provide the package interface should be public; private otherwise. Packages are organized into libraries and engines. There are fewer than ten engines for the entire multi-million LOC system described by Thuy (1992).

Dependencies which cross component boundaries are a significant problem. A dependency exists when each of two or more classes cannot function without a full implementation of the other.

'... it is difficult and extremely costly to test a component really independently of the components it depends on. That implies replacing those components by stubs, for which the reliability is not guaranteed and for which the costs and delays are very high. It is on the other hand possible and acceptable to test a component when those it depends on have themselves been tested. [This can be done if there are no dependence cycles, so] there must be no large scale dependence cycle.'

Dependencies among components (at any level) complicate integration testing. Dependencies and side-effects are to be removed, if at all possible. If dependencies are unavoidable between components at a given level, they must be contained within a single higher level component.

### 5.4.4. Incremental integration

A detailed analysis of integration testing is presented by Overbeck (1994a). The approach does not consider specific techniques to identify sequences, state values, or input values. Instead, test patterns are offered based on the relationship between client and server classes with all possible variations on inheritance.

Two basic test suites are the building blocks of the approach. Class unit test (*self test*) is passed if for all methods of a CUT: (1) message input/output are correct as specified; (2) every server class of the CUT is used properly; and (3) all intra-class operation sequences are correct. The second basic test suite involves a client $X$ and server $Y$ ($X$ contract-test $Y$). $X$ passes the contract-test suite when: (1) $X$ passes the self-test suite; (2) $X$ always meets $Y$'s pre-conditions; and (3) $X$ uses $Y$'s output correctly.

Six cases for application of these two basic test suites are considered: (1) self-test of a base class; (2) self-test of a single subclass; (3) contract-test of two base classes; (4) contract-test of base client and subclass server; (5) contract-test of subclass client and base server; and (6) contract-test of two subclasses. The extent of the test suite varies in each case.

To integrate an arbitrarily large system, an order of integration must be established. The self-test and contract-test patterns are then applied according to the order of integration.

'... if we self-test every class in the system, contract-test every client–server relationship between one client and one server, and contract-test every client–server relationship between multiple clients of a single server, we know that the complete system works correctly.' (Overbeck, 1994a, p. 87).

The problem is to accomplish this with minimal reliance on stubs. Contract-test cannot be passed when the usage structure of classes is cyclic ($X$ uses $Y$, $Y$ uses $Z$, $Z$ uses $X$). Stubs are necessary to break a cyclic usage between classes, but increase the cost of testing. An incremental strategy is presented to deal with cycles and reduce the total number of test cases.

**Step 1**: Find a class, $C$, whose superclasses and server classes have passed self-test.
(a) Self-test \( C \).
(b) Contract-test all of \( C \)'s servers.
(c) Repeat 1, 1(a), 1(b), 1(c) until there are no more \( C \).

**Step 2:** If there are classes remaining there must be a cycle. Find a minimal set of classes, \( S \), which have passed Step 1 and which contains a cycle.
(a) For each client in \( S \), contract-test it with all unit tested servers.
(b) If there are cycles in \( S \), find the smallest set of classes that could break the cycle. Develop and unit test stubs for each such class.
(c) Test the stubbed class set.
(d) Replace the stubs with the original classes and contract-test the resulting client and servers.

Systems tested by this process assure that:
'... all classes have been self-tested, all client–server relationships between two classes have been contract-tested, and all client–server relationships between multiple clients and single server have been contract-tested. Assuming the pre-conditions for self-testing and contract-testing, such systems are considered to be correct.' (Overbeck, 1994a, p. 116).

**5.4.5. Integration by propagation patterns**

An abstract model of class dependencies can be used to identify sets of objects needed for integration (Lieberherr and Xiao, 1992). The specification is an abstract representation developed to facilitate reuse and evolution. It consists of propagation patterns and class dictionary graphs (CDG). The CDG defines relationships among classes: inheritance, aggregation and association. A propagation pattern defines a responsibility (Wirfs-Brock et al., 1990) which can be implemented by an 'infinite family of programs'. The propagation pattern acts on classes; the classes are obtained from the CDG. Implementations may be derived by traversing a CDG with a propagation pattern. As a system evolves, initial CDGs are refined. The incremental changes from one CDG to another are called an evolution history.

A limited kind of integration test plan may be obtained from these representations. It is suggested that progressively more complex integration sets be used.

'Each component [a class] in an evolution history is tested individually, but it may be possible to reuse inputs from previous components. We want to test a component incrementally, initially with very simple inputs [other objects which are part of or used by the class under test] by exercising only part of the propagation patterns. Then we make the [integration set] more complex until all the propagation patterns have been sufficiently exercised.' (Lieberherr and Xiao, 1992, p. 319).

A propagation pattern and the CDG are used to expand these sets incrementally. The testing approach selects subsets of the propagation pattern to create a test unit. This is a set of objects necessary to exercise some or all responsibilities of a propagation pattern. Each successive test unit is more inclusive. The collective step-increments for test units are called a growth-plan.

'A growth-plan consisting of a long sequence of slowly increasing partial class dictionary graphs is good. The reason is that in general the addition of a small number of classes and
edges requires a small amount of additional code to be tested. We try to test small incremental steps to avoid testing software in big chunks.’ (Lieberherr and Xiao, 1992, p. 338).

This method does not address determination of specific parameter values, the state of objects in the test unit, or the state of the class under test for specific test cases. Also, determination of expected results is not considered.

‘... we do not give a comprehensive method for finding good test units and how to find suitable test objects for them. It is a difficult task, even at the level of logical circuits, to find a “small” number of inputs that exercise all parts of a given circuit properly.’ (Lieberherr and Xiao, 1992, p. 338).

Two adequacy criteria are offered. First, a growth plan is complete if it includes all objects that may comprise a test unit in the growth plan. A minimally adequate test plan (a collection of growth plans) contains only the objects necessary to exercise the class under test.

5.4.6. Integration by automatic reverse engineering

‘In OO testing, a tester often needs to answer questions like “to test this class what other classes must be tested?”’ A program representation called the object relation diagram (ORD) can be used to answer such questions (Kung et al., 1993). This diagram contains a node for each class and an edge for each inheritance, aggregation (instance variable) and association (message parameters and friends). It is asserted that some order (of class integration) exists which can ‘eliminate’ the need for test drivers or method stubs. This order can be discovered by analysis of the ORD.

If the ORD is acyclic, a stub-less test order can be obtained from the ORD by a topological sort.2 But cycles frequently happen. A strategy for testing without stubs is outlined. It is asserted that all cycles must contain at least one association (a ‘proof’ for this is noted but not given; it is easy to imagine counter-examples of cycles without associations). The strategy calls for ‘eliminating’ one association from the cycle, but does not explain how this is accomplished. Having broken the cycle, bottom-up testing may be performed. When the bottom-up testing is complete, the association may be reintroduced and tested.

5.4.7. Mode machine

A specification-based integration test plan may be obtained from a global state model synthesized from component state models (Binder, 1994b). To perform class, cluster, and subsystem integration, a mode machine (a composite state model) is synthesized from class state models. The mode machine represents the collective behaviour of the components under test. Sequences of external event/response paths (end-to-end threads) are identified as a side-effect of mode machine synthesis. Every externally determined thread may be enumerated and tested. This provides a systematic means to develop an integration test suite. The same test generation technique used for classes is applied. Mode-machine

---

2 A topological sort is an ordering of a directed graph such that all predecessors of every node are listed before the node. It cannot be produced if there are cycles in the graph.
threads have direct mapping to use-case based testing, and provide the basis for a system-wide test coverage measure.

'The application state model helps with several hard testing problems: (1) identification of all end-to-end threads, (2) unambiguous determination of external sequences necessary to test a thread given a certain constellation of class states, and (3) unambiguous determination of external sequences necessary to place an object in a given state.' (Binder, 1994b, p. 12).

Each event/response thread in the mode machine should correspond to a use-case. If all use-cases have been tested, but some event/response thread remains untested then either: (1) a gap in the test suite exists; (2) the use-case model is incomplete; or (3) the system under test has an unspecified capability or a surprise. This is the same kind of information provided by running a specification-based test suite on an instrumented implementation.

5.4.8. Integration by increasing scope

Effective integration may be accomplished by successively focusing on a larger scope (Siegel, 1994). The functional scope of the system under test should be limited so that the entire cycle runs no more than six weeks. There are three main stages.

Stage 1. Class-to-class integration:
   (1.1) classification: design verification;
   (1.2) protocol: class contracts;
   (1.3) interface: client/server arguments;
   (1.4) synchronization: time related exceptions;
   (1.5) aggregation: collection and associations.

Stage 2. Architectural integration:
   (2.1) horizontal layer: interfaces across abstract and concrete components;
   (2.2) functional subsystem;
   (2.3) cluster: any collection of client–server classes.

Stage 3. User interface integration:
   (3.1) GUI

The corresponding process model for incremental integration testing is discussed in Section 8.

5.4.9. C++ integration

A three step process for C++ integration is outlined by Lee et al. (1994): (1) map the relationships among system components—inheritance, friends, message passing, function calls and the main program; (2) next, test components as follows: (i) root objects—those instantiated from a superclass with no friends which receives no messages from other classes, (ii) other classes that have ‘the fewest inheritance, friend and message passing relationships’ and do not participate in any recursive relationships. Repeat, choosing classes with higher coupling until all classes meeting this criterion have been tested; (3) test classes with recursive relationships. Begin with classes having ‘the fewest recursive relationships with other untested classes’. Stub the untested classes. Repeat until all recursive classes have been tested; (4) test the main program.
5.4.10. Planning and factoring

Following unit (class) testing, integration testing is a gradual assembly of all components in the system (Desfray, 1994). There are two primary considerations: design-for-testability and an explicit plan for integration test. The plan provides both a time line and a technical order of integration.

The test strategy relies on a comprehensive set of design principles to avoid or remove mutual dependencies among classes. No cycles are permitted in graphs of class relations. Such relationships are to be factored into interface classes. When classes must communicate, a mailbox pattern (an event class) can be used to decouple recognition and response.

Planning is important since the point when a component becomes available may not coincide with a strictly technical order of integration. This may require the use of stubs (‘emulation’). When a stub is replaced, retesting is needed. Similarly, subclasses can only be developed and tested after their superclass(es) are tested. This constraint must be observed during integration.

5.4.11. Integration by composition

Conventional structural integration strategies do not work well for object-oriented systems (Jorgenson and Erickson, 1994). Without a ‘main program, there is no clearly defined integration structure. Thus there is no decomposition tree to impose . . . testing order of objects.’

‘The shift to composition (especially when reuse occurs) adds another dimension of difficulty to object-oriented software testing: it is impossible ever to know the full set of ‘adjacent’ objects with which a given object may be composed.’ (Jorgenson and Erickson, 1994, p. 33).

Integration can be accomplished by thread identification. A thread begins with an externally generated event (a user key press for example). In response, some sequences of messages are generated by objects. This sequence traces a path over the network of method interfaces in the system under test. The end of the sequence is an ‘event quiescence’ message, which typically causes some response to be sent to the system’s environment (a message is displayed to the user).

This end-to-end thread is an atomic system function (ASF). It provides a basis for integration testing by composition. The test approach requires identification and execution of ASFs. ASFs may be identified from a suitable representation or by analysis of source code.

5.4.12. Wave front integration

‘Wave front integration’ is advocated in the SASY methodology—see Section 8 (McGregor and Korson, 1994). This integration plan calls for an order of class development, so that system functions to be integrated are also the functions developed for a planned integration cycle. This avoids the need for stubbed methods. The need for good communication and planning is stressed, as is the importance of test automation.
5.4.13. Pragmatic application system integration

Object-oriented systems present greater challenges for integration testing than conventional systems. 'The functionality of an object-oriented program is distributed in a network of communicating objects and not in a top-down hierarchy of functions. Consequently, no easy top-down or bottom-up [test strategy can be taken from] method call relations.' (Jüttner et al., 1994c, p. 3). Two integration strategies are used: object integration and object communication.

'Object integration' is integration by uses (definition of class variable, message sending, etc). This requires code analysis. The steps are: (1) make an inventory and classify all uses in the system under test. A simple use is a use involving two objects; a complex use involves several objects. A cyclical use involves objects which use each other; (2) for each simple use, devise at least one test which exercises the use; (3) for each non-simple use, devise at least one test. Monitor simple uses covered by complex uses to see if any simple uses are tested as a side-effect; (4) for cyclical uses, break the cycle with a stub. Test the parts of the cyclic use separately, then test the whole cyclic use. This process continues until all uses have been tested.

'Object communication' is integration by relationships called out in an OOND model. As use-based tests are performed, message traces are recorded as an object communication diagram. This diagram is compared to the specified object communication diagram. Differences are used to devise additional test cases. Additional object-to-object relationships in the OOA/D model can be used to select more tests. For example, the QMT dynamic model can be used to select message sequences. Only one test case is developed for each such facet of the model.

5.5. System test

Object-oriented implementations do not obviate the motivation for system testing: to assess reliability and requirements compliance for an entire application system. Established practices for system testing can be transferred to object-oriented implementations with no loss of generality (Jacobson et al., 1992; Graham et al., 1993; Graham, 1994). These sources suggest using established system testing techniques (Myers, 1979; Beizer, 1984). In OOSE, system test should exercise at least the ‘basic course’ of all use-cases, all allocated requirements, and each operation described in user documentation (Jacobson et al., 1992).

A use-case based system test strategy that incorporates reliability optimization is used in the FREE approach (Binder, 1994b). The system test suite is derived from extended use-cases and event trace diagrams (also ‘object interaction diagrams’ or ‘fence charts’). A use-case is extended by adding an operational relation and estimates of relative usage frequency. Frequency estimation allows testing under the operational profile. This maximizes field reliability subject to available testing resources by testing most frequently used capabilities first (Musa, 1993). FREE supports either testing-to-cost or testing-to-MTTF. In a budget-constrained, test-to-cost project, test resources are allocated to operations in proportion to their relative frequency of use. In a reliability-driven, test-to-MTTF project, testing is conducted in accordance with the operational profile until an acceptably low failure rate is observed. Three levels of system testing may be performed: (1) 'use-case compliance' (extended use-case coverage); (2) 'reliability optimization' (level 1 under
control of the operational profile); and (3) 'integrity verification' (end-to-end thread coverage is obtained for a level 2 test suite).

5.6. Regression test

When a new or modified component has passed unit and integration testing, regression test strategies are concerned with selecting existing tests to repeat on other components, since a new or modified component may produce side-effects which cause unchanged components to fail. With the typically rapid pace of change in object-oriented development, some authors report that regression testing may be performed several times a day (Rettig, 1991; Love, 1992; Arnold and Fuson, 1994; Siegel, 1994). Simple changes may have significant impacts.

'Changing a class implementation, in general, requires that code be recompiled for classes that are derived from the changed class or which depend on its interface in any way. In a system that supports incremental loading of C++ code into a running program (as in a continuously running embedded system), any change to a class implementation will likely require that all extant objects of those classes be updated or replaced. (Coplien, 1992, p. 245).

The scope of a regression test would involve all these components. Within a class hierarchy, the basic scope of retesting established by Perry and Kaiser (1990) is that superclass changes warrant retesting of subclasses. Some testing of inherited methods need not be repeated (Harrold et al., 1992). McCabe et al. (1994a) argue that extensive regression testing may be unnecessary when the 'safe' strategy has been followed.

5.6.1. Regression test selection

An automated approach to selecting a class and application regression test suite is presented by Rothermel and Harrold (1994). The approach does not identify tests or a coverage per se, but does identify tests to be rerun after a class is changed. A safe regression suite reruns 'all tests that could possibly exhibit different output when run on [the system with changed class(es)].' A precise regression suite does not rerun any tests that cannot cause different output. No test suite can be both safe and precise since there is no way to determine, a priori, exactly which tests will pass or fail for the changed system.

Program dependence graphs are generated from the class or system under test. These graph models represent code segments, control and data dependencies among code segments and conditions which control branching. They also represent regions which define the conditions necessary to enter a conditional code block. The test model operates on the following entities:

- $P$ the source code for some version of a program
- $T$ the test suite for $P$, comprised of test cases
- $G$ the program dependence graph for $P$
- $H$ a region history, which lists all the test cases in $T$ which caused code in each region to be activated
- $P'$ the source code for a subsequent version of program $P$
- $T'$ the regression test suite for $P'$ comprised of test cases to rerun
With \( P, \ T, \ G, \ H \) and \( P' \) in hand, the test case selection strategy is straightforward.

Step 1. Generate \( G' \) from \( P' \).

Step 2. Compare \( G \) and \( G' \) to compose \( T' \). If there are modified regions or regions affected by other changes, use \( H \) to select test cases that previously exercised these regions.

Two variants of the program dependence graph are used: the interprocedural program dependence graph (IPDG) and the class dependence graph (CLDG). The IPDG is used to select \( T' \) for ‘applications programs’. The CLDG is used to select \( T' \) for modified or derived classes. Identification of the test suite for a class hierarchy requires two additional steps. A ‘representative driver’ is posited for the CLDG. This acts as a kind of main program for the class under test. Derived classes are flattened to account for interaction effects with base class features.

Polymorphic calls are not expanded in the graph. When a polymorphic call is encountered, all tests associated with the node containing the polymorphic interface are selected. It is argued that since the program dependence graphs represent data dependencies, encapsulation will tend to limit the extent of regression test. The approach is code based and will work with either white-box or black-box test cases.

5.6.2. Recursive support for regression testing: TOBAC

The capabilities of the TOBAC system (Siepmann and Newton, 1994) are discussed in Section 7. The conventional distinction between unit, integration, system and regression test are ‘blurred’ in object-oriented development. In some projects, ‘regression’ testing may be conducted daily. Object-oriented systems are typically composed of complex objects, each of which is a composite obtained by recursively expanding object instance variables. Since test cases must correctly initialize variables, the state of a complex object is very important.

‘Defining test cases for complex objects requires first and foremost the ability to create these complex objects and set them to specific states. There are three ways to do this: (1) use only existing methods, (2) defining special creation methods (constructors) only used for testing, (3) building a complex object from its subobjects.’ (Siepmann and Newton, 1994, p. 155).

‘[If a new method is added, it may produce states which are] inappropriate for the existing methods. Therefore every change to an object’s implementation may require complete retesting of that object.’ (Siepmann and Newton, 1994, p. 156).

Regression testing requires ‘fresh’ complex objects. If ‘canned’ objects are used to set the system state in a regression test, the effect of changes may be missed or masked.

TOBAC can construct and store complex objects to facilitate regression testing. The ODL (object description language) used in TOBAC provides a means to create new objects every time a test is rerun. ‘Maintaining a description of how to create a complex object is one of the keys for regression testing of object-oriented software.’ (Siepmann and Newton, 1994, p. 157) A replacement list is kept in a testcase object. Subclasses may be tested from a testcase for superclasses by using the replacement list. Regression testing of a class which has been specialized from a tested class is facilitated. Suppose a test
suite is developed which includes class X. Later, subclass X' is specialized from X. The replacement list allows X' to be tested with the original test suite.

6. TESTABILITY

6.1. Overview

Testability is a result of two factors: controllability and observability. A controllable implementation can be set to a desired state, have a desired set of inputs applied and be invoked in this state. An observable implementation does not obscure the occurrence of a fault or hinder determination of correctness. There are many ways that either property can be enhanced or diminished.

'Testability is the relative ease and expense of revealing software faults. . . . Software testing adds value by revealing faults. It is fundamentally an economic problem characterized by a continuum between two goals. A reliability-driven process uses testing to produce evidence that a pre-release reliability goal has been met. Time and money are expended on testing until the reliability goal is attained. This view of testing is typically associated with stringent quantifiable reliability requirements. Other things being equal, a more testable system will reduce the time and cost needed to meet reliability goals.

A resource-limited process views testing as a way to remove as many rough edges from a system as time or money permits. Testing continues until available test resources have been expended. Measurement of test adequacy or system reliability are incidental to the decision to release the system. This is the typical view of testing. Other things being equal, a more testable system provides increased reliability for a fixed testing budget.' (Binder, 1994d, p. 87).

Some kind of testability criterion is implicit in nearly all testing techniques. For example, Poston (1994) discusses what is needed to make an OMT specification 'test-ready'. As a practical matter, many testing techniques do not scale well. For example, the number of tests required in path covers like that of Parrish et al. (1993a) increases with the square of the number of operations in the class under test. (Lack of scalability may not be an indictment of a testing technique as much as it is of the developer's failure to manage complexity). Testability can be viewed as the practical limit imposed by any given test technique.

In this section, views that explicitly consider testability are summarized. Non-uniformity in exported domains and encapsulation leaks reduce testability (Freedman, 1991). Irregular subtyping is an obstacle (Leavens, 1991; McGregor and Dyer, 1993a). Lack of Demeter compliance can increase testing difficulty (Lieberherr and Xiao, 1992). The negative impact of complexity is described by Taenzer et al. (1989) and Ponder and Bush (1994); there are collaborating reports by Wilde and Huitt (1992), Lejter et al. (1992) and Li and Henry (1993). Inter-component dependency is seen as the single greatest obstacle to testability (Lakos, 1992; Thuy, 1992; Desfray, 1994). Software process is argued to be equally important (Binder, 1994d).
6.2. Uniformity

6.2.1. ADT testability criteria

A formal investigation of testability for abstract data types defines testability as the extent to which the IUT is controllable and observable (Freedman, 1991). These properties may be quantitatively assessed by analysis of the domains of input and output variables and their types.

An input inconsistency occurs when the same input does not always result in the same output. Some internal state variable cannot be controlled by the test case. An output inconsistency occurs when a member of the domain of an output variable cannot be produced. This is a lack of observability. ‘A domain-testable program does not exhibit any input–output inconsistencies.’ (Freedman, 1991, p. 561).

Global or local state variables reduce observability of procedure or function output, since response is not determined by input explicitly passed to the IUT. A function or procedure that cannot produce every possible value in the domain of its associated type is not controllable, since part of the output domain cannot be produced under any circumstance.

Observability may be improved by adding explicit arguments to replace access to global or local state variables. Similarly, they can be made perfectly controllable by defining a data type that exactly corresponds to the domain of the output variables. The number of changes that must be made to obtain domain-testability is an index of testability.

6.2.2. Monotonic inheritance

If the hierarchy of the CUT conforms to monotonic inheritance (Choi and Mannino, 1991) superclass test cases can be reused to test subclasses (Overbeck, 1994a). This requires: (1) that pre-conditions for subclass methods are the same or weaker than their superclass pre-conditions; (2) that post-conditions for subclass methods are the same or stronger than their superclass post-conditions; and (3) that every method must observe the class invariant and the invariant of its superclass(es).

This result eases formal verification of subtypes (Leavens, 1991). A simulation function is developed that maps a subtype to its supertype. If the mapping is complete and the supertype has been proven correct, then the subtype has been proved to be correct and will respond correctly to polymorphic messages. The requirements for the simulation (a mathematical formalism) are quite similar to the requirements for strict inheritance.

Monotonic inheritance allows pre-conditions to perform effective checking of polymorphic messages (Meyer, 1992). That is, if the hierarchy has been constructed so that subclass invariants are only the same as or stronger than the superclass invariants, then one can be confident that the subclass will respond correctly.

Some implications of ‘strict inheritance’ for testing are discussed by McGregor and Dyer (1993a). Testability is improved when states are disjoint and subclass states are disjoint partitions of single super-states. In the absence of strict inheritance, classes will require more testing and may be difficult to understand. Superclass test cases will be less reusable. It will not be possible to rely on assertions to detect incorrect client usage in a polymorphic server.
6.3. Complexity

The 'yo-yo' problem that results from the combination of inheritance, polymorphism and the self mechanism are discussed in Section 3 (Taenzer et al., 1989). It can be difficult to make such systems controllable. The TOBAC testing tool was developed in part to solve this problem (Siepmann and Newton, 1994).

But the 'yo-yo' problem poses a more fundamental obstacle to controllability. The technical ability to control a system is of limited value if it is not clear what needs to be tested. Several studies of object-oriented software maintenance have found understandability is often the first victim of object-oriented development (Lejter et al., 1992; Wilde and Huitt, 1992; Wilde et al., 1993; Li and Henry, 1993; Ponder and Bush, 1994).

An effective balance between good use of the powerful capabilities of paradigm and design-for-testability remains an open question. A general answer is probably not possible. There are many schools of thought on good design. Many of these attributes are listed in the implementation part of the testability 'fishbone' (Binder, 1994d). Figure 10 lists these attributes. Some of these implementation facets can be quantified (Lieberherr and Holland, 1989; Chidamber and Kemerer, 1991 and 1994; McCabe, 1993).

Several design principles can enhance C++ testability (Moreland, 1994): (1) avoid multiple inheritance; (2) minimize private inheritance with private data members; (3) use shallow inheritance hierarchies; (4) minimize inter-class coupling; (5) maximize intra-class cohesion; and (6) maximize hierarchic cohesion (implement true sub-type relationships).

![Figure 10. Implementation testability factors (Binder, 1994d)](image-url)
6.4. Dependencies

Dependencies among object-oriented software components arise for many reasons. Some are particularly error-prone and present significant obstacles for testing (Lieberherr and Holland, 1989; Dorman, 1993; Overbeck, 1994a; Jüttner et al., 1994a, b).

6.4.1. Architectural testability

In large systems, dependence cycles are the greatest obstacle to testability (Thuy, 1992 and 1993). A dependence cycle occurs when, for example, class P uses class Q, class Q uses class R, and class R uses class P. Either all participants in the cycle must be 'big-bang' tested, or at least one must be stubbed. With dependence cycles, 'there is no obvious order to testing at the class level.' If a cycle exists between classes, it may be possible to limit it to member functions only. Stubs are necessary when testing any implementation with cycles. Avoiding large scale dependence requires a good architectural design.

'The object-oriented approach by itself does not guarantee structural testability. Nothing prevents a designer from creating crowds of mutually dependent classes. . . .

We allow dependence cycles within a package, but we forbid them between packages and between any higher level modules. Side-effects (via global variables), cycles of side-effects and abuse of friend functions created considerable obstacles to testability.'

Aside from poor technical design, cycles may reflect the structure of the application problem:

'Mutual dependence is sometimes deeply rooted in the concepts of a domain. Geometry is a good example. In this domain we have elementary concepts such as point, line, circles, transformations, and more complex ones such as Bezier curves or Nurbs curves. A point can be constructed with two or three coordinates. It can also be constructed as the intersection of two curves, or as the result of a transformation. Thus the concepts of point depends on all the other geometric concepts. The same reasoning applies to all the geometric concepts: they all depend on one another. Directly mapping concepts into classes would indeed lead to a very poor design.'

Several design strategies are suggested for such situations. The general solution is to factor out a logical intersection of domains, such that the primary domains can share the intersection without circularity.

In large systems, classes must be organized into packages (subsystems). Five guidelines for testable C++ packages are given: (1) a package has a public interface composed of public classes; (2) all other classes in a package are not usable/visible outside the package; (3) all class implementations in the package are private; (4) the package interface is defined by abstract classes; (5) each client of the package must provide its own package interface (an adaptor) which is a concrete class derived from the abstract package interface.

6.4.2. Levelizable classes

Identification and elimination of cyclic dependencies is key for testability in C++ systems (Lakos, 1992). 'Cyclic dependencies among most all modules will occur if not carefully prevented, as will compile-time dependencies.' (Lakos, 1992, p. 276). Principles that enhance VLSI testability may be applied to class structures.
A collection of [classes] with an acyclic dependency graph is said to be 'levelizable'. The notion of level is borrowed from the field of digital, gate-level circuit simulation. The level number indicates the longest path from the gate to the primary input. Level-1 gates are fed by primary inputs, level-2 gates are fed by level-1 gates and zero or more primary inputs. This means no feedback. This is exactly the property we would like our design to possess.' (Lakos, 1992, p. 278).

As testing moves to higher levels of abstraction, tests exercise the interaction among components which comprise the IUT. Ideally, a test driver should be able to exercise a single component (at any level of abstraction) and only the components on which it depends. Cycles make this hard to do.

Several examples of C++ classes with dependencies and possible corrections are provided. The general approach is to identify the common element or programming practice which causes the dependency and to relocate it to a separate class. A protocol class may be useful. A class is factored into its interface (no data members and only pure virtual functions). The implementation is derived from this class.

Subsystems are composed of packages of classes. Packages should also be acyclic and provide an abstract interface that effectively hides the package details.

6.4.3. Inter-class dependencies

Simple inter-class uses can reduce testability (Dorman, 1993). Suppose one wishes to test class P4, given the inheritance hierarchy in Figure 11. Class P2 uses an object of class Q2, and P4 is derived from P2. To test P4, not only must its superclasses (P3, P2, P0) be completed, but one must also have Q2 and its superclasses (Q1, Q0) in hand.

'In practice, hierarchical class testing will often result in having to wait for the successful completion of other unit tests before a class can be completed. . . . A major consequence of hierarchical class testing is thus to lengthen the overall timescale of a project. In reality, as

![Figure 11. Inter-class dependency (adapted from the work of Dorman, 1993)]
a result of the complex inter-relationships that exist in any object-oriented design, hierarchical class testing is impractical in any non-trivial object-oriented environment. (Dorman, 1993, p. 77).

6.4.4. Lack of factoring

Testability is important to the success of integration testing (Desfray, 1994). Components must be unit tested first, i.e. ‘big-bang’ integration will not work. If classes are properly factored during design, high testability will follow.

Four design strategies for removing dependencies are suggested: (1) generalizing a property. Suppose two subclasses use each other. The feature of interest may be represented in a superclass replacing the dependency between two subtypes with an attribute of a common supertype; (2) decomposition into general and specific parts. Suppose C1 and C2 use each other and are thus cyclically dependent. The dependency can be removed if the C1 features needed by C2 are represented by a subclass of C1, say C11. Likewise the C2 features used by C1 are represented by subclass C22. Then C1 uses C22, C2 uses C11, and the dependency is removed; (3) relationships can be factored into interface classes which represent the facts dependent on the existence of two other objects; (4) when classes must communicate, a mailbox pattern (an event class) can be used to decouple recognition and response.

6.5. Process issues

Binder (1994d) argues that there are six main facets of testability: representation, implementation, built-in testing, test suites, test tools and process capability.

‘A usable representation is necessary to develop test cases. Implementation characteristics determine controllability and observability. Built-in test capabilities can improve controllability and observability and decouple test support from the application features. An adequate and usable test suite (test cases and plan for their use) is necessary. Test tools are necessary for effective testing; high leverage is available with an integrated tool set. Without an effective organizational approach to testing and its antecedents, technical testability is irrelevant.’ (Binder, 1994d, pp. 89–90).

7. TEST AUTOMATION

7.1. Overview

The systems discussed in this section automate some aspect of the testing process. They include proprietary tools developed for use in a single organization, research systems or tool proposals. The focus here is on how these tools can solve testing problems. It is not a detailed catalogue of features. The testing strategies underlying these tools are discussed elsewhere in this report.

Commercial ‘computer-aided software testing’ (CAST) products are not discussed here. At least a dozen CAST systems for object-oriented representations and implementations were available as of the fourth quarter of 1994. Product information is readily available from vendors and periodically surveyed by several organizations. CAST has limitations in testing large systems (Arnold and Fuson, 1994). Product-specific demonstration reports
may be found in the work of McCabe and Watson (1994) and Poston (1994). A taxonomy of capabilities useful for debugging is presented by Purchase and Winder (1991).

7.2. Automatic test generation/execution

7.2.1. FOOT

The FOOT system would ‘search’ and ‘parse’ a class to generate a test suite, execute it, and report the results (Smith and Robson, 1992). No details of the proposed approach are provided. A test driver class (‘test object exerciser’ or TOE) would be generated for each test run. The TOE declares an object of the CUT. A predefined test strategy (see Section 5.2.3) is used to produce messages to be sent to the OUT. The TOE runs test cases by sending the generated messages to the object under test. The results of the test are saved in a file. The user designates the CUT and the strategy to be used.

7.2.2. Test and debug tools

General test automation is needed for object-oriented development (Graham et al., 1993). Useful capabilities include flow of control tracing, the ability to select code paths and to set inputs, message content and state; to report state of the OUT and repeat tests. An object inspector is needed; three approaches are suggested: use get methods, make private features public or use a direct access debugger.

7.2.3. ASTOOT

ASTOOT produces test cases from an algebraic specification for a CUT (Doong and Frankl, 1994). A test case consists of two method sequences. Each sequence should result in the same state (sequences which should produce unequal results may also be generated). One sequence is generated from the left side of an axiom, the second from the right. These sequences are result-equivalent with different operation sequences. Since axioms are expressed in terms of other axioms, each sequence can be expanded to an arbitrary length by substituting other axioms (‘term rewriting’). If a right-hand side axiom contains a predicate (if-then-else), a branching tree results. Sequences that result from a predicate require the user to determine parameter values that will satisfy the predicates. The user may specify the length of expansion for sequences.

The ASTOOT drivers are automatically generated since all ASTOOT tests use the same pattern (Doong and Frankl, 1994). The driver reads in a test case (two equivalent sequences), checks syntax, sends sequences, compares returned objects using an equality operation built-in to the CUT and compares the result to the test case ‘tag’. The tag is the expected test case result: equal or not equal.

7.2.4. MKTC

The MKTC test suite supports C++ data scenario testing (Turner and Robson, 1993d) with three components: MKTC, MKMFTC and TESTRUN (Turner and Robson, 1992). MKTC generates test cases and a test driver given a description of data-member states (a ‘test script’). Development of the test script is discussed in Section 5. The MKTC
tool: (1) generates and lists test cases produced from the script; (2) generates a driver class; (3) allows replacement of individual test cases; (4) checks test script syntax; (5) optionally traces the test case generation; (6) simulates execution of the generated test suite; and (7) reports state initialization requirements to the user. The generated test cases may be interleaved with other user tests. MKMFTC (Make MakeFile for Test Cases) generates a make file for the components in a directory. TESTRUN runs the driver and logs output for user inspection.

7.2.5. Testgraphs

The Testgraph approach (see Section 5) generates test cases by traversing a state model of the class under test (Hoffman and Strooper, 1993a, b). Test messages are hard-coded for small classes but are selected by traversing the Testgraph model.

7.3. Test drivers

There are three kinds of test drivers. In a client–server driver, the driver (the client) declares an object of the class under test (the server). The driver sends messages to this object. The C++ friend function can be used in a similar manner. Client–server test driver class hierarchies are often constructed to mirror the hierarchy of the class(es) under test (Cox, 1988; Rettig, 1991; Love, 1992; Taylor, 1992; Wilke, 1993; Dorman, 1993; Firesmith, 1993b; Beck, 1994; Murphy et al., 1994; McGregor and Korson, 1994). In a subclass driver, the driver class is defined as a subclass of the CUT (Murphy et al., 1994; D’Souza and LeBlanc, 1994; Desfray, 1994). In a registration pattern, an object under test registers with a driver object, allowing the driver to send test messages to it (Firesmith, 1993b; Giltinan, 1994). A wide range of techniques for sending test messages are used, ranging from hard-coded message statements to fully dynamic interpreters that ‘execute’ a test script.

7.3.1. ICpak 201 test environment

A suite of test tools was developed to test the Stepstone Corporation’s ICpak 201 commercial class library. The application class hierarchy had a parallel TestTool class hierarchy, as shown in Figure 12 (Cox, 1988). The driver for each subclass was able to exercise all inherited methods for its target subclass as well as those specialized or refined.

The name of the class under test was provided as a parameter to TestTool. This allowed an object of any class to be instantiated as the object under test (OUT). Test cases were hard-coded in TestTools as messages sent to the OUT. The response to the test messages was evaluated in an assertion which defined the test pass/fail condition. If the test failed, the assertion provided uniform reporting of the failure.

A later report describes additional capabilities (Love, 1992). A session recorder automatically logged all developer interactions. The log could be ‘played back’ to reproduce failures occurring in an interactive session. A test driver generator would automatically generate a ‘test class’ driver for each class, and a ‘test method’ for each method. ‘The result was a parallel hierarchy of class tests that mirrored the original inheritance tree.’ (Love, 1992, p. 192). The method tester used random number generators to produce argument values. Method testers would check for memory leaks and correct type on returned items.
7.3.2. Smalltalk test manager

The Test Manager is a Smalltalk class that uses hard-coded test cases (Rettig, 1991). The test class responds true or false (pass/fail). Several modes may be designated: write a log file, display results in a window or start debugger. The user enters \texttt{TestManager testClass: ClassUnderTest}. If all tests pass, the driver answers 'true'. All such class tests may be invoked from a single \texttt{Test Procedure} class. This was used to perform daily integration testing.

7.3.3. Test jigs

To support 'thorough' testing before integration, including '... limits testing, coverage analysis and all the other metrics we normally apply to entire applications,' a dedicated \texttt{test jig} must be created for each class (Taylor, 1992).

'This jig is responsible for ensuring complete coverage of all the object's methods and data, confirming that each operates properly under normal conditions and fails gracefully under abnormal conditions.' (Taylor, 1992, p. 17).
The jig and all the associated test cases are to be stored with the class, under configuration management control. The jig must be consistent with the class. Every change to a class requires a complete retest using the jig. The jig is modified as needed.

7.3.4. ACE

The ACE testing tool generates a test driver from a user-prepared test script. The generated driver sends messages to the CUT and can compare actual and expected results for equality (Murphy and Wong, 1992; Wong, 1992; Murphy et al., 1994). The script has three parts: (1) compile and link; (2) declaration of variables and functions to set up the test; and (3) test cases. The test cases consist of a message trace and an expected result, which is the expected value of an object or variable and exception.

A test driver class is generated from the script, compiled, and linked with the CUT. The driver may be either a client/server or subclass driver. A subclass driver is used when the CUT has private methods or methods accessed by inheritance. Use of a subclass driver must be specified in the test script.

The tool is routinely used, but has some limitations. Results can only be evaluated for equality. This is not useful or feasible in all cases. The client/server configuration is not well suited to all class clusters. For example, it is impractical to use this kind of driver with GUI (graphical user interface) classes. Scripts cannot be embedded within a script.

7.3.5. Drivers from Z

A C++ test driver class and test cases can be automatically produced from a Z specification (Jia, 1993). The driver contains functions which map from the specification to implementation domains. The driver and the IUT are compiled together to produce a harnessed implementation. The harnessed implementation reads the test cases, tracks predicates exercised, and reports pass/fail results.

7.3.6. Cantata

Cantata is a commercially available tool that supports testing of C++ classes by generating a test harness and a complete set of stubs for the class under test (Dorman, 1993). The test harness is generated by parsing an input file containing parameter values corresponding to a test case. The code produced is a main program which sends and evaluates each test message. The system also generates stubs for all the server classes of the class under test. The stubs are minimal. This stubbing allows a developer to focus more directly on the interface and functionality of the class under test.

7.3.7. CASE tool

The test process for a distributed, multi-user CASE system used a test class structure that paralleled the application class structure (Wilke, 1993). This system supported structured analysis on a Sun-3 and was implemented in Objective-C. Unit testing was done by developing a test class for each application class, with a test method that corresponded to each application method.

This arrangement gave a very clear structure to the tests, and identified where the tests for a particular method would be. This meant if a method interface changed, the corresponding
changes to the test class were quickly identified. It was also possible to define test-classes in a hierarchy mirroring the real class hierarchy, with test methods being redefined [to match the application classes]. The testing arrangement was therefore quite complex but provided a very thorough testing procedure.’ (Wilke, 1993, p. 322).

7.3.8. Portable GUI

A test harness for C++ was used to test a reusable, portable GUI environment for an airline scheduling application (Wilke, 1993). Classes for the GUI environment were designed to be platform independent. Each class was tested separately.

‘Testing individual classes was relatively straightforward. Generally a test harness was built to exercise each of the classes used in an application. Classes were designed to be independent of one another. There was no notion of a state machine within each class, which would ultimately lead to increased coupling between classes, thereby reducing the potential for reuse of those classes.’ (Wilke, 1993, p. 314).

7.3.9. Duplicate and mixin hierarchies

Two strategies are useful for test drivers (Firesmith, 1993b). A hierarchy of test drivers can parallel the hierarchy of the class(es) under test, or a mixin class may be used if multiple inheritance is supported. A mixin class is not intended to be instantiated. Instead, it supplies features when it is inherited. A mixin may be used to avoid changing the class under test for test purposes (e.g. by adding state reporter methods). A TestClass is defined which inherits from both the class under test and the test operation mixin. TestClass is exercised by a driver.

7.3.10. Subclass driver

A driver implemented as a subclass allows access to encapsulated features of a CUT (D'Souza and LeBlanc, 1994). This avoids development of built-in testing, using debug probes or trusting untested accessor methods. The subclass driver does not require any modifications to the CUT, but allows full access to its implementation. Parameters passed to the CUT must be instance variables of the driver. Test cases are hard-coded in the example driver. The driver is called from a user interface class, and accepts a parameter that controls the number of tests to be run. The CUT and arguments are re-initialized for each test. See Section 5 for the associated test strategy.

7.3.11. Registry pattern

The registry pattern requires that each class contain a registration method (Giltinan, 1994). A Test Repository class is globally visible to all objects in the SUT. It contains a pointer to every object which has registered. Upon creation, an object sends its id to a test control object. The test control object may then send messages to registered objects, i.e. test cases. Upon destruction, an object de-registers. The test control class provides a user interface for testing, issues test sequences and can support queries on registered objects.
7.3.12. *iipt*

*iipt* is a Unix test tool for C++ (Parrish *et al.*, 1994). The testing strategy is to generate sequences of modifier messages. The tool avoids lengthy generation sequences by copying an object whose state has been obtained by a desired sequence as the next object to test. The tool allows selection of a number of messages in a sequence, produces some basic run statistics, and attempts to prune duplicate states. Input for the messages is produced with a random number generator. See Section 5 for the associated test strategy.

7.3.13. *Smalltalk TestSuite*

A simple Smalltalk test driver can be constructed with three classes: TestCase, TestSuite, and TestResult (Beck, 1994). TestCase instantiates the object under test, sends a message for the desired test, evaluates the result and then performs clean up. TestSuite contains a collection of TestCases and records the result of running each TestCase in TestResult. An actual test driver is created by subclassing each of these classes to test the class of interest. It is suggested that all application classes respond to the message `testsuite`, returning a collection of unit tests for the class.

7.3.14. *SASY*

A test driver hierarchy is developed which parallels the application class hierarchy in the SASY methodology (McGregor and Korson, 1994). Drivers are to be specialized from an abstract superclass to provide consistent logging and display of results. The drivers define and initialize the OUT and send a sequence of messages. Drivers 'should be able to determine whether the OUT has returned the appropriate value and placed itself in the appropriate state.' Ideally, tests are represented as unbundled scripts to promote reusability.

7.3.15. *TOBAC*

TOBAC is a test support tool for Smalltalk (Siepmann and Newton, 1994). The testing philosophy is discussed in Section 5. The system addresses the problem of setting a 'complex object' to a known, useful state. A complex object is the entire set of objects implied when a particular class is tested. Since objects are composed of other objects, which in turn are composed of objects, and so on, this is a non-trivial task. Every object involved in a test (arguments, instance variables, class variables, response and exceptions) may need a 'recursive' initialization. The system supports four phases of test execution: (1) initialization—global and class variables; (2) complex object creation—recursively set the state of all members of the complex object; (3) execution—set argument state and send a message to the OUT; and (4) evaluate the resultant state of the complex object to the expected state (this process is referred to as the 'Oracle' phase, but no means for producing the expected results is discussed).

The tool is implemented by five main classes: (1) `ObjectCreator` allows interactive definition of a complex object. The `ObjectCreator` results may be saved. The saved results are expressed in a representation that can be refined for regression testing (the TODL language); (2) `TestOracle` determines whether the object under examination meets expectations. The expected and actual complex object may be compared for equality, identity, no change (equal and identical), partial change, true, false, nil. Each object which fails
the comparison is reported; (3) **TestCase** performs creation, execution and comparison by sending messages to the OUT. **TestOracle** is invoked for each instance variable, response and argument; (4) **TestSuite** contains a list of testcases. Individual testcases can be used in more than one test suite; and (5) **TestCaseBrowser** provides a user interface for controlling test execution. It displays categories, classes, test suite, class hierarchy and test cases.

### 7.4. Oracles

An oracle is a means for producing the expected results for a test case. This is often difficult and is typically performed manually. Nearly all the object-oriented approaches surveyed rely on manual generation of expected results. Several approaches support comparators: automatic determination of pass/fail if the expected results are available. Some reports (Siepmann and Newton, 1994) refer to such comparators as oracles.

Object-oriented programs pose some unique difficulties for the production of expected results and evaluation of test runs. Comparing objects which represent expected results to objects which have accepted a test message may require a complex comparator. Class interface arguments are typically objects which have a more complex structure than simple character, integer or floating-point types. Equivalent abstract states may have different concrete states (Meyer, 1988; Doong and Frankl, 1994; Murphy *et al.*, 1994; Siepmann and Newton, 1994). Different objects of the same class may have identical states, so it is useful to be able to determine object identity in addition to object equivalence (Thielen, 1992; Davis, 1994; Siepmann and Newton, 1994).

Some authors argue that white-box testing techniques are self-sufficient, implying the implementation provides a kind of oracle (Smith and Robson, 1992; Kung *et al.*, 1993 and 1994; Parrish *et al.*, 1993b; Harrold and Rothermel, 1994; Siepmann and Newton, 1994). That is, if the code ‘works’ for a given test suite, the implementation passes. At best, code-based testing may produce exceptions or output that are *prima facie* evidence of a fault: memory leaks, attempts to use a missing feature, deadlock, livelock, runaway loops, hardware or operating system interrupts (overflow, underflow, divide by zero, etc.), resource allocation errors or grossly inadequate performance. In trivial applications, it is often possible to assess test pass/fail by *ad hoc* inspection. Allowing these exceptions, there is no non-subjective means to decide the correctness of results produced by white-box test cases which is not a tautological fallacy.

The Testgraph approach requires that the tester devise and program a state model of the CUT (Hoffman and Strooper, 1993a, b). The model is limited to certain states of interest, but can produce the expected results as each element of a test sequence is executed.

Some automated specification-based approaches use self-checking tests as oracles. The equality relation in an algebraic specification means that the exact state of the CUT need not be given in advance; it suffices that two instances of it are ‘observationally equivalent’ after each has accepted an equivalent but different message sequence (Doong and Frankl, 1994). The test results produced by the CUT are substituted in constraints expressed in a model-based specification (Jia, 1993). If the CUT produces results which are within the specified bounds, then the test is passed.
7.5. Built-in tests

Built-in tests refer to features of a class designed to facilitate testing, apart from any other features the class provides.

7.5.1. Gauges

The gauge concept uses built-in tests to promote reusability (Cox, 1990). A specification language is composed of test procedures. A test procedure takes a single argument that designates the implementation to be tested. The test procedure runs the implementation to see if it complies with an associated specification, or ‘gauge’. The test procedure and the specification are separate items. This has three implications: (1) it ‘could’ decrease the importance of source code for understanding to allow reuse and change, and ‘could’ allow other ways of locating components; (2) it ‘could’ facilitate greater language interoperability; and (3) it ‘can provide rigor to open-universe situations when compile-time checking is not available.’ (Cox, 1990, p. 32).

7.5.2. Assertions

Assertions can be used to document and implement a specification, to provide built-in testing, and to implement defensive programming. The Objective-C ‘assert’ macro is suggested as a ‘rudimentary’ specification and testing language (Cox and Novobilski, 1991). Assertions are used for ‘white-box’ testing, and grouped into pre- and post-conditions. They should be used to ‘verify that the conditions the method depends on were met by the caller. For example, verify that argument values and/or types are as they should be for the method to work properly.’ Post-conditions ‘verify that execution achieves the expected result’ (Cox and Novobilski, 1991, p. 98). These kinds of assertion provide a ‘particularly cogent and readable kind of documentation.’ (Cox and Novobilski, 1991, p. 99).

Assertions may be combined with trace capabilities. If assertions are placed in constructors and destructors, the entire life of an object may be traced (Thielen, 1992).

Assertions and exceptions are useful but not sufficient as built-in tests (Firesmith, 1992 and 1993b). They are of limited use for self-testing with concurrent processing, and ‘may not detect problems immediately.’ However, they respect encapsulation, impose relatively low overhead, and have a straightforward implementation.

Assertions are built-in to the Eiffel language (Meyer, 1988) and implement ‘design by contract’ (Meyer, 1992). They are a deliverable in the Fusion methodology (Coleman et al., 1994) and the Class Relation methodology (Desfray, 1994).

Several types of assertion are used in conjunction with a registration driver (Giltinan, 1994). The Testlnstance method verifies internal consistency by checking the class-invariant and post-conditions. The MakeSafe method turns on pre-condition checking following construction, class-invariant checking before destruction, and heap integrity checking post-destruction. The IsSafe method checks the class-invariant and returns a boolean.

C++ classes should not ‘... process garbage input [instead they should] recognize and flag it to the programmer as early as possible’ using range checks on input and assertions (Davis, 1994). A related problem is dealing with an incorrect object address. ‘Non-virtual methods invoked with an invalid object pointer are called normally; however, the pointer
is incorrect and the members accessed from within the method are not correct. The solution is to provide a non-virtual checking member function.

7.5.3. State manipulation

Several sources suggest or require the addition of methods to set, reset, or otherwise manipulate state (Turner and Robson, 1993d; Siepmann and Newton, 1994; Binder, 1994d). This improves testability by increasing controllability, but can pose encapsulation and security hazards (Firesmith, 1993b; Binder, 1994d).

7.5.4. Equality operators

Tests may be designed so that pass/fail can be evaluated by comparing the object(s) under test for equality to another object of the class (Murphy et al., 1994; Doong and Frankl, 1994). This is attractive since it is readily automated and is appropriate for many kinds of test. Manual inspection of a pair of objects can be difficult and error-prone. An equality operation is implemented in each class to be tested. The definition of object equality may be fine-grained or approximate: e.g. comparing every entry and pointer in a list, or simply checking the length, head and tail. If the equality method uses other methods in the CUT, a ‘buggy’ method could give a false report. Equality is irrelevant for some audio and video outputs and cannot directly check time and sequence dependent behaviour.

7.5.5. Object identity

Testing may require the ability to determine object identity (Thielen, 1992; Davis, 1994; Siepmann and Newton, 1994). Different objects may be identical in state and structure, but participate in different associations. Identity must be evaluated in some situations. Several C++ coding practices can make the object identifier explicit and observable (Davis, 1994). For example, an accessible object identifier may be created by the constructor and accessed by a public member function. The TOBAC tool can automatically generate tests to compare actual and expected object identity (Siepmann and Newton, 1994).

7.5.6. Concurrent/self tests

Concurrent built-in tests (BITs) allow objects to perform runtime self-test by asynchronous communication with concurrent built-in test processes. This approach is expensive and only warranted in high-reliability applications (no implementation details are provided). Self-testing objects accept test messages, perform a self-check, and report status. Self-checking BIT poses several problems: (1) the OUT does not initiate the self-checking (it may not be performed when necessary); (2) the self-checking methods can provide a trapdoor; and (3) the self-checking method can violate encapsulation if it modifies the state of the OUT (Firesmith, 1992 and 1993b).

7.5.7. VLSI strategies

Built-in tests are necessary to provide controllable and observable objects (Binder, 1994d). Four BIT strategies are described, following established practices in hardware
design-for-test (DFT): (1) no DFT—this approach will result in the highest possible test cost or least reliability, depending on which basic testing strategy is followed; (2) ad hoc DFT—this results in a relatively high test cost or less reliability, depending on the basic testing strategy. Class testing is done by developing a driver for each class and possibly a set of stubs. This improves controllability and observability. However, the absence of explicit set/reset can result in lengthy setup sequences and reliance on debug probes to verify state; (3) structured/standardized DFT—the ad hoc strategy is augmented by including a standard BIT application programmer interface (API) in all classes, i.e. set/reset, state reporting and BIT-safe implementation. A single driver must be developed for each class. To develop the driver, the CUT interface is hard-coded in the driver. Test cases must be manually prepared for each class/cluster. The structured/standardized strategy for class testing calls for a driver to be paired with every class. There is additional development and maintenance effort for every class. With n classes, there are n additional interfaces to synchronize, and limited automation of test case generation, execution and evaluation; (4) advanced DFT—a single, generic driver would solve these problems by generating a test suite from an embedded test specification (t-spec). It would activate the CUT with this test suite and evaluate the results automatically. The need for a driver paired with every class is obviated.

7.5.8. Message sequence checker

If a regular expression which represents the sequential constraints of a class is stored in a class variable, objects of the class can perform runtime checking of incoming message sequences (Kirani and Tsai, 1994). The Sequence Checker is based on established algorithms for detecting equivalent finite state automata. If regular expression specifications are used throughout a class hierarchy, then the Sequence Checker can also evaluate sequences which contain superclass messages. The specification and testing strategy for the Sequence Checker is discussed in Section 5.3.

8. SOFTWARE PROCESS

8.1. Overview

The software process is 'the set of tools, methods, and practices we use to produce a software product.' (Humphrey, 1989, p. 3). A test process is the set of tools, methods, and practices which have an explicit role in the design and execution of test suites. With object-oriented development, the test process is closely intertwined with all other process facets (Siegel, 1992; Jacobson et al., 1992; Taylor, 1992; Pittman, 1993; Graham et al., 1993; Siegel, 1994; Graham, 1994; Binder, 1994d; Desfray, 1994; McGregor and Korson, 1994). The focus here on testing issues should not be interpreted as suggesting that other process facets are any less important.

There are several primary issues to consider in devising an effective testing process of object-oriented implementations.

(a) What is an effective test and integration strategy given the iterative approach preferred for object-oriented development?
(b) When should testing begin?
(c) Who should perform testing?
(d) How can testing techniques help in preventing errors?
(e) How are testing activities integrated into the software process model?
(f) How can testing facilitate reuse?
(g) To what extent should reusable test suites be considered a necessary part of a reusable component library?

This section presents generic process approaches by date of publication. Each experience report in the following section also reflects a specific process.

**8.2. Process proposals**

**8.2.1. Code and test**

The first process model for object-oriented development that considers testing appears in the work of Shlaer and Mellor (1988). This is essentially a linear phased approach which simply indicates that code and (unit) test are to be followed by integration and acceptance. No other aspects of testing are discussed.

**8.2.2. Three roles**

Testing is performed by three primary actors: class implementors, class designers, and a test and integration team (Booch, 1991). Test process policies and standards are to be devised by the test and integration team. Class implementors do unit testing, ‘in which individual classes and modules are tested.’ Class designers perform functional subsystem testing. The test and integration team assembles ‘compatible subsystems into a complete subsystem and runs system-level tests.’

Development of reusable ‘test scaffolding’ (drivers and instrumentation) and test suites is recommended for each kind of testing. ‘This approach builds confidence in the lower level abstractions of the system first, in terms of its key abstractions and mechanisms.’

**8.2.3. Collaborative test strategy**

The iterative and incremental nature of object-oriented development is well-suited to a test-as-you-go approach (Siegel, 1992). This blurs the typical distinctions between detailed design, programming and unit testing. Complete coverage of method statements and ‘limited path coverage for object interactions’ is advocated. Several novel test design and integration approaches are proposed (see Section 5).

A kind of double-teaming is advocated where two developers work in tandem on a single class. One concentrates on design, development and the execution of tests. The other concentrates on implementation design and programming; hence the ‘collaborative’ rubric. This allows test cases to be prepared and executed as small increments of code are completed. Significant improvements in productivity are reported.
8.2.4. OOSE testing strategy

The object-oriented software engineering (OOSE) approach covers development from initial concept definition through system installation (Jacobson et al., 1992). Testing is considered to be a necessary part of OOSE. The test strategy follows generally accepted practice and describes techniques for unit, integration, system and regression testing. Early test planning is advocated (the \textit{V} model); a battery of white-box and black-box test case design techniques are to be used and organised around the \textit{use case}. The use case is a key concept in OOSE. ‘When a user uses the system, she or he will perform a behaviorally related sequence of transactions in a dialogue with the system. We call such a special sequence a use case.’ The OOSE test strategies are discussed in Section 5.

8.2.5. Parallel test and application development

A test driver and regression test suite should be developed for each class; each class should pass all tests before being integrated, and functional testing should be performed on sub-systems and systems (Taylor, 1992). This process requires that every component be ‘thoroughly’ tested before integration, including ‘... limits testing, coverage analysis and all the other metrics we normally apply to entire applications.’ A dedicated test ‘jig’ must be created for each class. Every change to a class requires a complete retest using the jig. A similar approach is to be taken for application subsystems. ‘Conventional black-box’ testing should be performed on the entire application.

8.2.6. Test effort model

Testing plays a significant role in an incremental (as opposed to iterative) approach to object-oriented development (Pittman, 1993). A design-code-test process is followed for some small slice of the total requirements (a slice is a subset of capabilities of the system under development). This is repeated until all the slices have been developed. The scope of the final \textit{acceptance test} includes all slices to complete the project.

The Cocomo model (Boehm, 1981) for the development of conventional data processing systems (‘organic mode’) allocates between 42\% (small project) to 36\% (large project) of the total budget to programming and unit testing. System and acceptance test allocation ranges from 16\% to 25\%.

Pittman argues that ‘system-acceptance test for object-oriented development is typically substantially shorter than in a conventional development, provided the components are of sufficient quality.’ (Pittman, 1993, p. 45). Phase effort estimates for object-oriented development are derived by adapting Cocomo for slice-wise incremental development. Half of the total test budget is allocated to acceptance testing. The other half is spread equally over preceding slices. A representative project budget for incremental development is shown in Table XVIII.

8.2.7. Test procedures with recursive development

The tasks to be performed in unit and subassembly testing are arrayed in a linear process (Firesmith, 1993a). Components requiring recursive (iterative) development are deferred. The steps in non-recursive development include design, coding and testing to create subassemblies and systems. Inspections, configuration management activities, search
Table XVIII. Object-oriented testing effort: Cocomo extrapolation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Percentage of total effort by phase and project size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small</td>
</tr>
<tr>
<td>Requirements</td>
<td>16</td>
</tr>
<tr>
<td>slice 0</td>
<td>12.8</td>
</tr>
<tr>
<td>slice 1 to n</td>
<td>3.2/n</td>
</tr>
<tr>
<td>Design</td>
<td>26</td>
</tr>
<tr>
<td>Programming and unit test</td>
<td>42</td>
</tr>
<tr>
<td>Test</td>
<td>16</td>
</tr>
<tr>
<td>Integration test for slice 1 to n</td>
<td>8/n</td>
</tr>
<tr>
<td>System test for slice n</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: The system is produced in \( n \) increments (slices).

for reusable components and updating system documentation are required at keypoints. Once a stable set of components is present, prototyping and further recursive development proceeds.

8.2.8. Test case schema for object-oriented programs

A test plan for object-oriented implementations requires several facets unique to object-oriented implementations (Berard, 1993). A summary of the items in the schema is presented in Figure 13. The schema is a tree, so there will typically be many branches and leaves for a single root (object test case). The schema is consistent with IEEE/ANSI standard 829 for test plans and test cases.

‘In testing objects, we are concerned with more than just “input parameters.” We must be concerned with the state of the object being tested, and quite possibly the states of other objects in the same system. In addition, we recognize that the testing of even a single operation in the interface to an object may involve a number of distinct steps.’ (Berard, 1993, p. 284).

8.2.9. V-model for object development

The V-model\(^3\) applies to object-oriented development, but there are significant differences (Graham et al., 1993). Unit testing must take account of the fact that a class is a bundle of functions. Testing is most effective when taken in a certain order: top down in a class hierarchy (superclass first, then subclasses). The order for a class should be constructor, accessor, predicate, modifier and destructor. Hidden features are to be tested first, since these are typically used to respond to public methods. Intra-class state-based testing is suggested to perform intra-class integration. Black-box and white-box tests are recommended in the work of Turner and Robson (1993a).

Inter-class integration should proceed in a bottom-up order of uses, testing classes

\(^3\) The V-model refers to a software process where there is a corresponding test activity for each activity that produces representations and implementations: concept definition drives acceptance test, requirements definition drives system test, architectural design drives integration test and detailed design drives unit test. A graph of this process resembles a ‘V’ (Myers, 1979; IEEE, 1986).
which have few or no suppliers in the system under test, then selecting classes with
tested suppliers, and so on, until the root of the uses hierarchy has been covered. This
strategy requires the use of drivers for nearly every class.

8.2.10. Class certification

Reuse requires high reliability, so reliability modelling of software components is
useful (Wohlin and Runeson, 1994). This process is appropriate for an object-oriented
component library.
'During development for reuse, a usage model must be constructed in parallel with the development of the component. The usage model is a structural model of the external view of the component. The usage model describes the possible behavior with no probabilities having been assigned to the events. The probabilities of different events are then added, creating a usage profile, which describes the actual probabilities for these events.'

The approach is an application of random testing strategies and the operational profile. The process has six steps: (1) model software usage; (2) derive the usage profile; (3) generate test cases for the profile; (4) execute test cases and collect inter-failure data; (5) certify reliability; and (6) predict future reliability. The State Hierarchy model (SHY) is used to represent usage and to derive an operational profile for a component. Probabilities are based on a Markov model of state transitions. The process uses a statistical model for stop-testing decisions based on cumulative failure rate and time spent testing.

8.2.11. Iterative/recursive/parallel development

A four phase test process is outlined in the work of Siegel (1994). The phases constitute an iteration, which produces a coherent slice of the system under development in four to six weeks. An iteration is subdivided into four mini-cycles, each lasting one to two weeks. These cycles are: structural, alpha, beta and iteration.

The structural cycle deliverables include a complete description of all files and classes to be used, stubbed methods with trace (as needed), first pass at class hierarchy models and a test plan. The test plan includes class unit test plans, a class verification plan, a preliminary integration test plan and preliminary integration test data.

The alpha cycle deliverables include preliminary implementations for class methods to be produced in this iteration, related error/exception handling, a revised integration test plan and test cases and a preliminary plan for testing concurrent and asynchronous processes.

The beta cycle produces the first roll-out of the iteration functionality. Deliverables include regression tests for interfaces, regression tests for aggregations and a fully developed test plan for concurrent and asynchronous processes. Integration testing is performed across all architectural layers and the user interface, as needed.

The iteration cycle completes the current iteration and sets the stage for the next iteration. A full regression test is to be run on all other components in the system under development. Key deliverables include evaluation of reusability and a schedule for the development of missing, incomplete or immature components in a subsequent iteration.

An automated test environment that can build and test the system 'nightly' is recommended. This requires standard names, library design and conventions, macro design, strict compliance with work product requirements, a single person in control of the process and diligent recording and tracking of defects.

8.2.12. Managing risk by comprehension

Managing the risk and reward of testing requires a balance between comprehension (knowledge of the reliability of the system under test) and testing effort (McCabe and Watson, 1994). Comprehension is needed to judge the relative quality and reliability of the SUT. Testing provides information, but at a cost. Metrics can help to point to components and interfaces that may be error-prone. The Object-S metric is claimed to measure '. . . system integration complexity from the point of view of selected classes,' but no operational definition is provided.
Visualization of class relationships can also guide testing effort. Tools can display flattened ('collapsed') classes or 'bound' classes. Dynamic binding is shown by lines that correspond to all possible bindings. This allows a kind of simulation: for each dynamic binding, one can be selected and the results observed.

The Safe strategy (see Section 5) provides a balanced approach. Integration testing can be minimal—exercise each method at least once—or maximal—exercise every possible call pair at least once—or a compromise. In addition to metrics and complexity charts, qualitative factors can be used to choose a compromise. Table XIX compares these strategies.

8.2.13. SOMA methodology

Testing is an explicit part of the 'semantic object modelling approach' (SOMA) methodology (Graham, 1994). Object-oriented development is argued to offer high(er) quality since it facilitates both testability and reuse. The modularity and specification/implementation correspondence inherent in object-oriented systems improves testability; reuse is a kind of re-testing.

An organizationally separate testing group is recommended. Testing is to be integrated with prototyping, design and programming. Specific guidance for test case design is not provided. 'Task scripts' provide a framework for test case identification. A task script is similar to the 'use case' of Jacobson et al. (1992).

'... every sequence of mouse clicks is a potential path [through the code]. The solution is to use the task scripts to represent potential paths through the system. If possible, paths should be grouped into equivalence classes that can be tested as one path. ... Follow all the usual, good, conventional testing practices ... State matrices and decision tables remain useful.' (Graham, 1994, p. 340).

Side effects can result when inheritance violates encapsulation. Other problems with inheritance indicate that 'the entire [class] structure should be integration-tested top-down.' A 'test harness' is suggested. This is a regression suite based on task scripts. It '... must automatically test through all inheritance levels. Do this at least as soon as prototypes are released to users.'

<table>
<thead>
<tr>
<th>Table XIX. Covers for polymorphic binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration test strategy</td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>Optimistic</td>
</tr>
<tr>
<td>Pessimistic</td>
</tr>
<tr>
<td>Balanced</td>
</tr>
</tbody>
</table>
The main steps of the SOMA test process are: (1) prototype; (2) white-box test one class; (3) integrate library classes; (4) black-box test; (5) publish user-interface documentation; (6) train users; and (7) conduct user acceptance testing. Prototyping is seen as a kind of specification test. Specification-based testing may fail to reveal implementation-specific faults (e.g. exception handling), hence ‘white-box’ class testing is advocated. Following class integration, ‘black-box’ system testing can focus on ‘... unexpected effects of the system working as a whole’. Integration should ‘test servers prior to client objects, because defects in servers may be propagated to their clients.’

8.2.14. Class relation methodology

In the class relation methodology, ‘it must be assumed that an untested class does not operate correctly.’ (Desfray, 1994). Test definition, test development, test execution and debugging are necessary. Correctness is compliance with invariants, pre- and post-conditions: ‘The more precisely the invariants pre- and post-conditions are defined, the less important test verifications become.’ (Desfray, 1994, p. 289). The hypergenericity representation is a low-level, language-independent specification which allows a test class to be automatically generated from a test case schema.

Unit tests test classes. No specific test case design techniques are discussed. Test cases are to be expressed in a formalized schema. A test driver is developed (or generated) for each class. This driver is implemented as a subclass of the object under test. The driver cannot modify the OUT and must be able to call private and protected methods. Abstract classes are tested by a driver subclass, which is an instantiation of the abstract class. The subclass driver needs privileged access to the OUT. This is possible via a subclass in all languages except C++, which must use a friend for the driver.

Integration testing follows class development. Key considerations for planning integration are cluster feasibility (the complexity of a class cluster is an upper limit), testability (components must be unit tested first and free of cycles), schedule (the dates on which the components must be available for testing) and cost. The technical details of this approach are described in Section 5.

8.2.15. Fusion

A test process for the Fusion methodology is sketched by Coleman et al. (1994). Testing object-oriented systems differs from conventional testing in ‘levels of abstraction, object state and inheritance.’ Assertions are advocated. The need to retest inherited methods in subclasses is also noted.

No details are provided for translating a Fusion model into a test plan, but a short checklist for the implementation of test cases is provided. Each class test should include five generic tests: (1) exercise set/get method pairs; (2) exercise associative operations; (3) verify initialization. In C++, constructor/destructor consistency should be checked. In Eiffel, default feature values should be inspected for correct content; (4) check C++ implementations for safe casting; and (5) attempt to trigger all exceptions.

8.2.16. SASY

The ‘software architect’s synthesis’ (SASY) process model is a collection of established techniques for object-oriented representation and implementation (McGregor and Korson,
1994). Verification and testing play a prominent role. Testing is considered to include static techniques (incremental reviews and inspections) as well as exercising code. Manual approaches to model validation are advocated. These include defining usage scenarios, hand simulation, and review by 'experts'. Checklists for completeness and consistency among work products are presented. Work products for testing are to be produced along with representations and implementations. Design for testability is emphasised in most deliverables.

'... testing is continuously interwoven into the development process. This not only locates faults early, it makes subsequent phases less likely to create new faults based on existing ones. This assumes an iterative incremental approach in which the steps of the development process are repeated and the product is developed through successive refinements.' (McGregor and Korson, 1994, p. 61).

Code testing is to be managed for efficiency. Several strategies are identified to support this goal, without giving details. A 'strict inheritance' design criterion is followed to promote testability and correctness (see Section 5). Reuse of member function test cases under inheritance follows the scheme of Harrold et al. (1992). Test cases are primarily 'functional', following the minimal state-based testing coverage suggested by McGregor and Dyer (1993b). Client-server contracts are suggested as an additional source of test cases, and orthogonal arrays are suggested as a means of selecting tests of polymorphic servers, but no details are provided.

9. EXPERIENCE REPORTS

9.1. Stepstone ICpak201

A well-defined test process was used to develop the Stepstone ICpak 201 (Cox, 1988; Love, 1992). This was one of the first commercial class libraries, providing reusable Objective-C classes for the development of GUIs on a wide range of platforms. The library contained 50 classes, 1200 methods and required 800 pages of documentation. Testing was conducted by a small group with limited testing resources. The testing group was independent of the development group.

'The testers were responsible for determining the specification that each class claimed to implement [by] interviewing the implementors and crosschecking the documents and the code. Then, they wrote a separate class, a TestTool, that exercised the implementation.' (Cox, 1988, p. 45).

Each application class was supplied with a 'demo' designed to exercise features of the class. The demo programs were run so that the Unix tcov utility could count the number of times each method was activated. Heavily used methods were excused from further testing. This avoided 'the need to write additional test cases for 900 methods.' (Love, 1992, p. 191). The remainder were reviewed. If a method was judged (subjectively) to have less than a 50% chance of containing an observable fault, then it was excused from further testing. 255 methods were tested, and 'more than 25%' of those tested contained observable faults.

Testing was done with the assistance of several tools. A session recorder was developed
to capture all *ad hoc* interactions with a CUT. The log could be ‘played back’ to reproduce failures occurring in an interactive session. The time intervals of events could be changed allowing a kind of ‘fast forward’. The log could be edited and replaced, allowing test suites to be developed easily. A test driver generator was developed that would automatically generate a ‘test class’ for each library class, and a ‘test method’ for each method (see Section 7 for details of this approach). ‘The result was a parallel hierarchy of class testers that mirrored the original inheritance tree.’ (Love, 1992, p. 192).

These drivers sent messages to the MUT. Where possible, random number generators were used to produce argument values. Method testers would check for memory leaks and correct type on returned items. Tests ‘... for inheritance were exhaustive: for each class, it was verified that every method [from] all its superclasses was executable and gave correct results.’ A dispatcher class integrated all test drivers and the recorder. It selected the appropriate log (from a previous session) for appending, checked the test setup, and logged the date and time of the test, the tester and the version.

The test process was designed to motivate the testers. They were asked to spend an hour a day trying to ‘break code’ by using long sequences. The testing of a method was complete when any one of five conditions was met: (1) all tests passed; (2) code reading okay; (3) adequate demo use; (4) low probability of an error; or (5) automatic test okay. Each method was required to pass at least one criterion before product release. Daily team briefings were held on progress which improved morale.

Only 7% of the code was still suspect after the demo-based frequency analysis. Forty per cent of the errors in this code were found by inspection or tests based on the inspection. Another 40% were found by automatic testing and the balance by ‘*ad hoc*’ testing.

The resulting class library had a very low in-service failure rate. ‘In the first year after ICpack 201 was shipped, only five reproducible errors were reported by customers.’ (Love, 1992, p. 195).

### 9.2. Medical equipment

An implementation-based unit test strategy for C++ programs used in an embedded medical equipment application is described by Fiedler (1989). The basis path technique (McCabe, 1976) was used to identify test cases for member functions, followed by domain analysis to assign specific values. Classes derived from ‘well covered’ base classes received less testing than those with an uncovered base. Tests of the ‘signals or exceptions that are raised (not propagated) by each function’ were also devised. Test cases were written to ‘ensure complete path coverage of each member function.’ Details of this approach are discussed in Section 5.2.

This technique revealed 5.1 defects per thousand lines of delivered source instructions. Fiedler (1989) concludes ‘the main difference we have found so far is that each object must be treated as a unit, which means that unit testing in an object-oriented environment must begin earlier in the life cycle.’

### 9.3. Apple Computer

Apple Computer decided to improve its software quality assurance (SQA) and test process when it began using C++ on the Finder project (Martin *et al*., 1989). The existing
SQA process used general procedures but was technically *ad hoc*. SQA performed alpha testing after a software system was complete or obtained from a third party for integration. The major SQA goals for the Finder project were to:

'. . . render the product more 'testable', to determine observable measures of testability, and to describe the material gain to the company that was realized. . . .[to] establish a more reliable and reproducible testing process that could be applied to other software projects . . . the new process was intended to use code incorporated in the product for testing purposes.' (Martin *et al.*, 1989, p. 130).

This would bring a shift from quality assurance to quality engineering by 'assuming a role in influencing product design according to some general principles for quality.' The SQA team became proficient in C++ with two months of training. SQA proposed testability guidelines for the developers, including minimizing the depth of the class hierarchy, minimizing overridden operators and methods, and minimizing the use of procedural C code.

'Corporate culture at Apple is a significant issue for its employees. That this project originated with SQA engineers and was supported by their managers and development engineers was recognized as an example of the Apple culture at work. (Martin *et al.*, 1989, pp. 138–139).

The most tangible result was in the development of some testing tools and built-in-test capabilities in application classes. The tools could list class hierarchy, method invocation cycles, graph method flow, count overridden operations, and count polymorphic functions. Other capabilities included an assertion mechanism, verifying successful completion and journaling the outcome of a test run. User-visible functions included: *Command* (controls the IUT), *Flag* (manages a set of state variables), *Verify* (receives messages from code instrumentation), *Screen* (interfaces with user), *Debugger* (interfaces to Macsbug), *Logger* (generates and stores logs). Logs were designed to allow rerunning test scenarios. The built-in-test code proved useful for field debugging. The test drivers could accept events (messages) from any possible client. This reduced testing time by 'a factor of ten.'

The team developed their own GUI driver. 'The interactive nature of using a Macintosh is a blessing to the user, but a bane to the tester.' The system was treated as a set of features—quality was not measured in terms of coverage. 'Improving source code was not the objective . . . assuring proper functionality was an objective.'

### 9.4. Network simulator

Testing is briefly mentioned in an experience report on using Eiffel to build a simulator (Gindre and Sada, 1989). No specific test design or coverage approaches are discussed. The process was incremental:

'The design-implement-test sequence is applied to the lifecycle of individual classes, but not necessarily to the lifecycle of the project as a whole. . . . The design, implementation and testing of classes in a cluster proceeds in a bottom-up fashion.'

Test cases were reused. 'Every potentially reusable class must be accompanied by: (1) documentation about how to use it and to inherit from it, (2) examples of use, (3) test sets.'
9.5. Team with test tool

Early test development and some simple testing tools facilitated high productivity and quality in a Smalltalk project (Rettig, 1991). The programming team followed a well-defined process in which testing played a major role. The test goals were to test every path in a method once, develop and use reusable test procedures, perform all testing under peer review and minimize the tedium of testing by following an ‘incremental glass-box’ strategy. A test plan was written and reviewed in tandem with class design.

A TestManager class was developed. TestManager responds to testing messages which direct it to ‘execute this block of code, report if response not as expected.’ TestManager has several output modes: log to a file, display on a window and start a debug tool. The reviewed class test plan was implemented as a set of messages which could be executed by TestManager. A class test was run by entering:

TestManager testClass: (ClassUnderTest)

‘When we complete a module and integrate the new code with the official version of the prototype, the TestManager is invaluable.’ A master procedure was used to check all classes. This allowed testing of the entire system at any point in development.

The team produced roughly 50,000 lines of Smalltalk. Programmers were responsible for their own testing (contrary to some recommendations) but test plans were written and reviewed using a detailed test standard. Table XX shows the effort used to produce test code and application code. Thirty per cent of the total effort went to design and test case development, 33% in coding, and 33% in test and integration. ‘... we have had less than a dozen errors slip through the testing process and appear in a subsequent module. In my experience, that is remarkable.’ (Rettig, 1991, p. 28). Similarly, Beck (1994) recommends 25–50% of a developer’s time be spent in unit testing.

9.6. Commercial Smalltalk

Some lessons learned while developing Smalltalk systems for commercial software products indicate that simple test process steps can be effective (Klimas, 1992).

‘Smalltalk programs permit classes to be used and tested as soon as they are designed and the initial underlying methods are defined. This permits meaningful testing to occur earlier

<table>
<thead>
<tr>
<th>Project activity</th>
<th>Percentage of total effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application development</td>
<td>code</td>
</tr>
<tr>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>comments</td>
</tr>
<tr>
<td></td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>coding</td>
</tr>
<tr>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Test automation</td>
<td>comments</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>TestManager</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Prototype</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
in the development cycle and offers the cost saving opportunity to diagnose and correct problems earlier. Therefore it is important to design Smalltalk programs for easy testability of the code from the start.' (Klimas, 1992, p. 2).

Simple guidelines, if followed, save time in later regression testing. Testing should begin as early as possible and allow incremental unit and integration testing. Each developer should prepare a formal unit and integration plan for the classes they produce. This plan should cover every method, and every method path. The scope of testing under inheritance should be set as per the criteria of Perry and Kaiser (1990). Unit test each method (instance and class) and integrate by project or application. Tests should try to reveal general classes of problems, not specific glitches. Try to cause exceptions during normal operation. Every class should be documented by a set of executable usage examples. Use an independent system test group to validate requirements.

Every class should implement a SelfTest method which runs a class test suite and returns true if all pass. Test results should be a character display for a screen or written to a file (not to a window, as a ‘buggy’ class may be unable to do this). Display a time-stamped log in a transcript window. Display application windows (momentary) during each test. Develop reusable low-level drivers (e.g. keystroke generation). Performance metrics should be captured and monitored for degradation.

One third of the total project budget should be allocated to testing. If the guidelines are followed, a reusable, executable test suite will accompany the application. This will reduce the cost of regression testing and raise its efficiency.

9.7. Testing in the huge

A defined and integrated test process was used at Matra Datavision for the development of CAD/CAM products supporting mechanical design (Thuy, 1992 and 1993). These are very large systems (thousands of classes, ‘several million lines’ of C++). This scale required several levels of integration testing, including class clusters, packages with tens of class clusters, and subsystems with tens of packages. Matra ‘has developed a complete and consistent object-oriented software development methodology.’ Testing and design for testability is stressed throughout the process.

Four main principles guided the approach: (1) analysis and design must integrate two perspectives—interfaces among operational subsystems and structural relationships among subsystem components; (2) analysis focuses on application semantics, design on architectural concerns; (3) several clearly delineated levels of abstraction are necessary—the lowest is the class. Packages, libraries and engines provide successively larger aggregations; (4) subsystems and their components must be independent, in addition to being modular and encapsulating.

The primary design-for-test strategy with architectural design is to maintain independence of subsystems and components, at all levels of abstraction. The details of the Matra approach are reported in Sections 5 and 6.

9.8. IBM EOSE system

The EOSE (Europe/Middle East/Africa order and supply execution) provides scheduling and fulfilment of IBM customer orders (Capper et al., 1994). This was ‘one of the largest
single business application developments ever undertaken by IBM.' Most functions are batch processed, with GUI for necessary user input. Development followed an 'informal OOD' approach, characterized as:

'... a traditional mainframe development project. ... It was recognized that a successful object-oriented development needs to be organized as an iterative process, but it also had to fit with the well-established waterfall approach to which the EOSE project management were committed.' (Capper et al., 1994, p. 136).

Prototypes were developed in ENFIN/2 (see the following experience report). C and PL/1 were used for implementation since no suitable object-oriented language was then available for the target environment. However, this code ‘... was structured according to object-oriented principles, using “home-grown” inheritance techniques.’

The test process followed iterative code development, and each code increment was ‘thoroughly tested’. No specific testing techniques are discussed.

‘Quality and code inspections were held, but only to check for consistency across objects and for conformance to standards. The continual testing demonstrates that the code functioned correctly. Completed client business objects were subjected to a full end-to-end test with the corresponding servers.’ (Capper et al., 1994, p. 137).

System testing was performed by an independent testing organization. Automated testing was used to verify conformance to requirements. Users subsequently performed acceptance testing using ‘real-world’ input. Quality metrics for this project are shown in Table XXI.

9.9. IBM SRFE query system

The SRFE (sales representative front end) system provides desktop access to sales and marketing data (Capper et al., 1994). SRFE uses object-oriented client processes running on an OS/2 workstation which communicate via an SNA LU 6.2 network to a remote site AS/400 database server. Development started in January 1991, with the first release being delivered in September 1991.

An ‘informal OOD’ process was followed. A requirements model was developed by prototyping and domain analysis using the CRC approach (Cunningham and Beck, 1986). Developers were assigned a related group of classes. Client components were prototyped and implemented in ENFIN/2, a Smalltalk-based object-oriented language and development environment. ENFIN/2 provides SNA communication services and a class library.

Table XXI. IBM object-oriented development metrics

<table>
<thead>
<tr>
<th>Project</th>
<th>KCSI‡</th>
<th>Effort, staff months</th>
<th>Developer test</th>
<th>Independent test</th>
<th>User test</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOSE</td>
<td>109</td>
<td>236</td>
<td>1.2</td>
<td>1.1</td>
<td>0.3</td>
<td>0.0</td>
</tr>
<tr>
<td>SRFE R1</td>
<td>19.4</td>
<td>72</td>
<td>N/A</td>
<td>2.1</td>
<td>0.46</td>
<td>0.0</td>
</tr>
<tr>
<td>SRFE R2</td>
<td>4.5</td>
<td>6</td>
<td>N/A</td>
<td>1.6</td>
<td>1.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

‡KCSI—one thousand changed source instructions.
Class testing was integrated with iterative class development. No details are provided about the test case design approach. Each incremental refinement was tested and integrated with completed classes. With each increment, an increasing proportion of clusters had been tested several times before being handed over for integration testing. The goal of integration testing was to ensure that each cluster "performed its responsibilities accurately."

"All of this testing had a cumulative effect on the final user acceptance testing because by this point many of the base modules of the application had been "in use" for some months." (Capper et al., 1994, p. 140).

Table XXI shows defect metrics collected for this project.

The test process incorporated GUI usability evaluation after requirements had been modelled and prototyped and again before the system was released. Cluster development (programming and unit testing) was tightly integrated and iterative. Integration test is a kind of mini-system test, with an external focus. Regarding the defect rates, the authors observe:

"In absolute terms . . . production code quality was excellent, for example, the zero defects of SRFE in over 12 months of production.

Further benefits were felt to be due to the increased code exercise [testing] from earlier in the development as a result of using an iterative approach. However, this approach did not obviate the need for traditional quality assurance activities (inspections, for example)." (Capper et al., 1994, p. 153).

9.10. Nuclear material simulator

A simple test process was followed to develop a simulator used to plan processing and storage of nuclear reactor fuel (Chao and Smith, 1994). It had three subsystems: event modelling, material modelling and application reporting. This system was written in 14300 lines of CLOS with 77 defclass constructs, 96 defgeneric constructs, 184 defun constructs and 227 defmethod constructs.

"The test approach is object-based; that is, the focus is on testing objects and their features." A different test strategy was used for each subsystem. Inspections were used to detect errors in requirements, specifications and code. The testing goal was to reveal component faults and component interaction faults.

Test scaffolding (drivers) was developed for each subsystem. Although the scaffolding was not itself tested, successful execution of the implementation under test was deemed to have verified the driver. Modules performing basic services were tested first, event data structure objects were tested second, and simulation driver objects were tested last. A greater 'coverage' (of an unspecified nature) was sought for the event modelling subsystem because its capabilities were used throughout the simulator.

Testability was improved by writing testable requirements, conducting inspections and the 'intuitive' nature of CLOS constructs. Class modularity and the resulting absence of side-effects eased fault diagnosis. However:

"A considerable amount of time and effort was required to generate comprehensive test cases because the partitioning of the software in different modules and packages was not obvious. A clear-cut software organization would help to expedite the generation of test cases." (Chao and Smith, 1994, p. 51).
The material modelling subsystem accounted for 70% of the application classes. Rela-
vitely less testing was performed on it because it was ‘infeasible to generate and execute
test cases for all combinations of classes and functions/methods.’ Interactions among
classes that represented independent problem entities were not tested.

An application subsystem was developed for each simulation model. This provided
reporting on the results of a particular simulation. No specific tests are mentioned;
however, these reports were used for ‘acceptance testing’. Observing that many questions
remain open, the authors call for attention to testing object-oriented implementations
comparable to that given to conventional testing.

9.11. Siemens AG

Integration plays a key role in an overall test strategy reported by a unit of Siemens
AG (Jüttner et al., 1994a, b). The strategy is simple but effective. Two main steps are
followed. First, intra-class integration testing is performed, with server objects stubbed-
out. The order of integration follows inheritance; base classes first, refined and newly
derived member functions second, followed by interleaved sequences of inherited, new
and redefined member functions.

Integration of classes in an application is performed with specification-derived test cases
applied to application threads. When a new class is integrated with an existing library, a
more extensive (but unspecified) coverage is sought.

The integration strategy promotes classes from unproven to proven one at a time.
Within the scope of the system under test, objects are identified as either proven or
unproven. Testing proceeds by testing each unproven object with trusted components.
When the tests pass, the OUT is added to the proven group. When an OUT uses other
unproven components, the unproven objects are stubbed-out to allow unambiguous results
for the OUT.

The same degree of testing is applied to modified classes as is performed for entirely
new classes. When only the implementation changes (and the interface and specification
do not), regression testing is facilitated by using the unchanged version to produce
expected results. Encapsulation is seen as helping, since side-effects are prevented.
However, inheritance is seen as increasing the complexity of test planning and execution.
Lessons learned from this experience have been presented by Jüttner et al. (1994c) and
are discussed in Section 5.4.

9.12. Mead Data Central

Testing object-oriented code running in a distributed, heterogeneous environment with
stringent performance and reliability requirements posed many challenges (Arnold and
Fuson, 1994). Mead Data Central (MDC) supports on-line computer-aided research for
legal, financial, medical, news and business topics. The system must provide rapid search
response, high availability (24 hours/day, 7 days/week except for a short daily down-time
window), accurate and repeatable results and timely print delivery. The target environment
includes IBM mainframe systems under the MVS operating system, many Unix systems
and desktop PC clients.

Testing was integrated with design and programming. Unit (class) white-box testing
was a developer responsibility. Early definition of public interfaces allowed early development of test drivers. Work on a class testing package began as soon as the class interface was defined. Completion of class development called for a test package in addition to application code. The test package included a test driver with embedded test cases and a file containing expected output. Successive class versions tended to have more similarities than differences. This eased automated generation and execution of regression tests.

The scope of unit testing was a C++ class or class cluster. No particular strategy for unit testing was reported. Class unit testing was relatively easy since useful test tools are available to support class testing. The tester can concentrate on specification compliance. Individual methods tend to have simple behaviour, so they are highly testable. Since a class interface was typically defined before its implementation was developed, test drivers were developed in parallel with the class implementation. Test drivers for higher level (base) classes could usually be reused with a few changes for lower level classes in the same hierarchy. Test drivers ‘grew’ along with the application classes, thus enabling iterative and incremental development.

Some aspects of unit testing were problematic. Testability was reduced by several factors: complex behaviours and component parts, attempts to reuse code outside of the contexts envisioned by the designer, and the use of non-OO software for implementation with side-effects due to diminished or non-existent encapsulation.

Upon completion of unit testing, several levels of integration testing were performed: (1) single Unix process; (2) intra-platform thread (e.g. cooperating Unix processes on a single machine); (3) inter-platform thread (e.g. PC/Unix/MVS). Integration tests were devised by choosing external inputs that corresponded to a specific thread of execution. Integration testing was difficult at all levels. The compound effect of differences among levels proved challenging. Available commercial test tools were not effective for large scale integration (thousands of components). Inconsistencies across class libraries were problematic and could only be dealt with by conventions enforced by design and code reviews. Problems noted included different approaches to encapsulation, and implementation of overloaded operators for collection classes.

The complex target environment caused significant technical problems. Many commercial classes were not thread-safe or re-entrant and control of resources under concurrency was not addressed or was incorrect. Platform specific debug and test tools hampered controllable testing and debugging of threads that crossed platforms (e.g. MVS and Unix). There is an impedance mismatch between object-oriented design and client/server architecture. Available client/server interfaces may be inconsistent with typical class architecture. The DCE (The Open System Foundation's ‘distributed computing environment’) and C++ exception semantics are different, so it was difficult to do controlled testing of exception handling.

9.13. Network systems management

TRACS is a network management system used to monitor traffic and outages on telephone central office switches connected by trunk lines (Murphy et al., 1994). The system has been in use for several years. It is implemented in Eiffel and C++. Both the application and the testing strategy have evolved over subsequent releases.

Detailed design, programming, unit testing and integration testing of clusters was done in parallel (a cluster is 'a collection of classes that relate to a common aim'). (Murphy
and Wong, 1992). Unit testing was performed on individual classes only if they: (1) were present in the OOA model; (2) performed file input or output; (3) performed asynchronous messaging; (4) were abstract or generic; (5) were 'complex'; or (6) were a component of a reusable cluster. Unit testing each class in an application-specific cluster was initially considered unjustified; testing was conducted on the cluster instead. Cluster-level test cases were derived from functional narrative requirements. Minimal integration testing was done. System testing was initiated when all cluster tests were completed.

Tests were executed by a proprietary automated tool (ACE) which allowed tests to be saved and reused for regression testing (Murphy and Wong, 1992; Wong, 1992). 'Each test case is described by providing a trace (i.e. a sequence of messages to objects) and associating with it some aspect of the expected behaviours of the class in response to that trace.' A test script contains declarations and initialization information accessible to all test cases. Some limitations were noted with the tool: it determined test pass/fail by comparing the object under test and a test object instantiated with the expected value for equality only. In some cases, there was no direct or meaningful implementation of an equality operation for the object under test. There was no direct way to deal with all the classes in a cluster as a single unit in the tool.

This initial approach proved to be problematic (Murphy et al., 1994). As clusters grew, many class interfaces were not exercised by the reused test suite, even though new tests were added to exercise new functionality. The focus on the cluster of classes lead to skipping code reviews. In turn, this allowed many 'garden-variety' defects (for example, using a variable before initialization) to escape early detection. Defects remained hidden in layers of clusters. The cost of late discovery and correction was very high compared to early discovery.

The process was revised to address these problems. Individual class unit testing was instituted, making use of a 'grey-box' approach. The test tools were refined to support class testing. An OOA specification was developed per cluster and used to derive more effective test cases. A complete cluster test plan was prepared (meeting nearly all the documentation standards in IEEE 829). These test plans were inspected. The changes provided significant improvements.

(a) On average, defects found in system testing were reduced by a factor of 70 to 1.
(b) Total development effort was cut by half.
(c) Latent defects in existing classes were revealed and corrected.

Process metrics indicated that the approach was effective.

(i) An average of five defects per class were revealed by automated class-level testing.
(ii) Of class development time, 5–30% was used for testing.
(iii) Of cluster development time, 5–20% was used for testing.

Overall, the TRACS experience suggests that established practice for an effective testing process is applicable to object-oriented development.

References
1. Surveyed sources


2. Related sources


## APPENDIX A: SOURCES, METHODS, LANGUAGES AND TOOLS BY SUBJECT

<table>
<thead>
<tr>
<th>Section</th>
<th>Subject</th>
<th>Source</th>
<th>Method</th>
<th>Language</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>ADT: formal verification</td>
<td>Guttag et al., 1978</td>
<td>Mod-Pascal</td>
<td>DTVS</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>ADT: formal verification</td>
<td>Olthoff, 1986</td>
<td>LARCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>ADT: formal verification</td>
<td>Leavens, 1981</td>
<td>SIMPL-D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Gannon et al., 1981</td>
<td>LARCH</td>
<td>DAIISTS</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>McMullin and Gannon, 1982</td>
<td>LARCH</td>
<td>CLU</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Choquet, 1986</td>
<td>LOFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Liskov and Guttag, 1986</td>
<td>Prolog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Bougé et al., 1986</td>
<td>Prolog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Gaudel and Marre, 1988</td>
<td>Prolog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Bernet et al., 1991</td>
<td>LOFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Jalote, 1992</td>
<td>SITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Zweben et al., 1992</td>
<td>RESOLVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Dauchy et al., 1993</td>
<td>PLUSS</td>
<td>LOFT</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Jia, 1993</td>
<td>Z</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>ADT: white-box testing</td>
<td>Liskov and Guttag, 1986</td>
<td>LARCH</td>
<td>CLU</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>ADT: white-box testing</td>
<td>Hayes, 1986</td>
<td>Z</td>
<td>Pascal</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>ADT: white-box testing</td>
<td>Parrish et al., 1993a</td>
<td>Ada</td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Cox, 1988</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Taenzer et al., 1989</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Cheatham and Mellinger, 1990</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Smith and Robson, 1990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Purchase and Winder, 1991</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Doong and Frankl, 1991</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Leavens, 1991</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Trausan-Matu et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Chung and Lee, 1992 and 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Jacobson et al., 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Tuy, 1992</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Berard, 1993</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Moreland, 1994</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Taenzer et al., 1989</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Leavens, 1991</td>
<td>LARCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Purchase and Winder, 1991</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Meyer, 1992</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Ponder and Bush, 1994</td>
<td>Smalltalk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Frankl, 1989</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Fiedler, 1989</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Smith and Robson, 1990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Subject</td>
<td>Source</td>
<td>Method</td>
<td>Language</td>
<td>Tool</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Jacobson et al., 1992</td>
<td>OOSE</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Hoffman and Strooper, 1993a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>McGregor and Dyer, 1993b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Turner and Robson, 1993d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Doong and Franki, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Kirani and Tsai, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Kung et al., 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Binder, 1994b</td>
<td>FREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Saikkinen, 1988</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Fiedler, 1989</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Thiel, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Reed, 1993</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Shopiro, 1993</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Cargill, 1993</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>D'Souza and LeBlanc, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Friesmith, 1993b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Reed, 1994a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Moreland, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Davis, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Fiedler, 1989</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Perry and Kaiser, 1990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Cheatham and Mellinger, 1990</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Trausan-Matu et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Love, 1992</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Jacobson et al., 1992</td>
<td>OOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Friesmith, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Thuy, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Harrold et al., 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Parrish et al., 1993b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Berard, 1993</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>McGregor and Dyer, 1993a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>McGregor 1994a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>McCabe et al., 1994b</td>
<td>Safe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Fault taxonomies</td>
<td>Purchase and Winder, 1991</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Fault taxonomies</td>
<td>Trausan-Matu et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Fault taxonomies</td>
<td>Smith and Robson, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Fault taxonomies</td>
<td>Friesmith, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Fault taxonomies</td>
<td>Hayes, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Automated model validation: formal</td>
<td>Kemmerer, 1985</td>
<td>Ina Jo</td>
<td>Pascal</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Automated model validation: formal</td>
<td>Gerrard et al., 1990</td>
<td>OBJ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Automated model validation: formal</td>
<td>Jones, 1990</td>
<td>Ada</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Dauchy et al., 1993</td>
<td>PLUSS</td>
<td>LOFT</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Selic, 1994</td>
<td>ROOM</td>
<td>ObjectTime</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Muller and Shlaer, 1994a and 1994b</td>
<td>Shlaer/Mellor</td>
<td>ObjectBench</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Fiedler, 1989</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Thuy, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>McGregor and Sykes, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Smith and Robson, 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Kung et al., 1993 and 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Parrish et al., 1993b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Turner and Robson, 1993c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Hoffman and Strooper, 1993a, b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Parrish et al., 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>D'Souza and LeBlanc, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Harrold and Rothermel, 1994</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Testing Object-Oriented Software

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>McCabe et al., 1994b, C++</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Overbeck, 1994b, Objective-C</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Cox, 1988, Objective-C</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Frankl, 1989, Eiffel ASTOOT</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Cob, 1990; Cox and Novobilski, 1991, Eiffel ASTOOT</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Doong and Frankl, 1991, 1992, Eiffel ASTOOT</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Jacobson et al., 1992, OOSE</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Loughman, 1992, OOSE</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Siegel, 1992, OOSE</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>McGregor and Dyer, 1993a,b, C++</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>McGregor and Korson, 1994, Objective-C</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Kirani and Tsai, 1994, Objective-C</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Poston, 1994, OMT</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Binder, 1994b, FREE</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Siegel, 1992, FREE</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Thuy, 1992, CDL</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Lieberherr and Xiao, 1992, FREE</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Jacobson et al., 1992, OOSE</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Kung et al., 1993, C++</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Desfray, 1994, Class Relation</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Jorgenson and Erikson, 1994, Objective-C</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Siegel, 1994, Objective-C</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Lee et al., 1994, C++</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Overbeck, 1994a, C++</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>McGregor and Korson, 1994, SASY</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Binder, 1994b, FREE</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Jüttner et al., 1994c, C++</td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Jacobson et al., 1992, OOSE</td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Graham et al., 1993, Objective-C</td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Graham, 1994, OOSE</td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Binder, 1994b, FREE</td>
</tr>
<tr>
<td>5.6</td>
<td>Regression test</td>
<td>Rothermel and Harrold, 1994, C++</td>
</tr>
<tr>
<td>5.6</td>
<td>Regression test</td>
<td>Stiepmann and Newton, 1994, Smalltalk TOBAC</td>
</tr>
<tr>
<td>6.2</td>
<td>Testability: uniformity</td>
<td>Freedman, 1991, Smalltalk</td>
</tr>
<tr>
<td>6.2</td>
<td>Testability: uniformity</td>
<td>Leavens, 1991, Smalltalk</td>
</tr>
<tr>
<td>6.2</td>
<td>Testability: uniformity</td>
<td>Meyer, 1992, Eiffel</td>
</tr>
<tr>
<td>6.2</td>
<td>Testability: uniformity</td>
<td>McGregor and Dyer, 1993a, Eiffel</td>
</tr>
<tr>
<td>6.2</td>
<td>Testability: uniformity</td>
<td>Overbeck, 1994a, Eiffel</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Taenzer et al., 1989, Objective-C</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Wilde and Huit, 1992, Smalltalk</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Leiter et al., 1992, C++</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Li and Henry, 1993, Classic Ada</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>McCabe, 1993, C++</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Ponder and Bush, 1994, Smalltalk</td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Moreland, 1994, C++</td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Thuy, 1992 and 1993, C++</td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Lakos, 1992, C++</td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Dorman, 1993, C++</td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Desfray, 1994, Class Relation</td>
</tr>
<tr>
<td>6.5</td>
<td>Testability: process</td>
<td>Binder, 1994d, C++</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Smith and Robson, 1992, C++</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Turner and Robson, 1992, Foot</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Jia, 1993, MKTC</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Graham et al., 1993, C++</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Hoffman and Strooper, 1993b, TestGraphs</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Doong and Frankl, 1994, Lobus, Eiffel ASTOOT</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Cox, 1988, Objective-C</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Rettig, 1991</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Wong, 1992</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Murphy and Wong, 1992</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Taylor, 1992</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Love, 1992</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Jia, 1993</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Wilke, 1993</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Wilke, 1993</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Dorman, 1993</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Parrish et al., 1994</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>D’Souza and LeBlanc, 1994</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Murphy et al., 1994</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>McGregor and Korson, 1994</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Siepmann and Newton, 1994</td>
</tr>
<tr>
<td>7.4</td>
<td>Oracles</td>
<td>Hoffman and Strooper, 1993a,b</td>
</tr>
<tr>
<td>7.4</td>
<td>Oracles</td>
<td>Doong and Frankl, 1994</td>
</tr>
<tr>
<td>7.4</td>
<td>Oracles</td>
<td>Jia, 1993</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Cox, 1990</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Thielens and Thielens, 1992</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Firesmith, 1992 and 1993b</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Turn of Robson, 1993</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Gilman, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Siepmann and Newton, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Doong and Frankl, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Murphy et al., 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Davis, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Siepmann and Newton, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Binder, 1994d</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Davis, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Kirani and Tsai, 1994</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Binder, 1994d</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Shlaer and Mellor, 1988</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Booch, 1991</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Siegel, 1992</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Jacobson et al., 1992</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Taylor, 1992</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Firesmith, 1993a</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Pittman, 1993</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Graham et al., 1993</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Berard, 1993</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Siegel, 1994</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Desfray, 1994</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Wohlin and Runeson, 1994</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Coleman et al., 1994</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Graham, 1994</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>McCabe and Watson, 1994</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>McGregor, 1994b</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Cox, 1988</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Gindre and Sada, 1988</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Martin et al., 1989</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Fiedler, 1989</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Rettig, 1991</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Murphy and Wong, 1992</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Love, 1992</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Thuy, 1992 and 1993</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Kimas, 1992</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Capper et al., 1994</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Chao and Smith, 1994</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Jutten et al., 1994a,b,c</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Arnold and Fuson, 1994</td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Murphy et al., 1994</td>
</tr>
</tbody>
</table>
## APPENDIX B: SUBJECT, SOURCES, METHODS AND TOOLS BY LANGUAGE

<table>
<thead>
<tr>
<th>Section</th>
<th>Subject</th>
<th>Source</th>
<th>Method</th>
<th>Language</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>ADT: white-box testing</td>
<td>Parrish et al., 1993a</td>
<td></td>
<td>Ada</td>
<td>dtest</td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Firesmith, 1992</td>
<td></td>
<td>Ada</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Jones, 1990</td>
<td></td>
<td>Ada</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Jia, 1993</td>
<td>Z</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Cheatham and Mellinger, 1990</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Thuy, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Moreland, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Fiedler, 1989</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Hoffman and Strooper, 1993a</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Turner and Robson, 1993d</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Kung et al., 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Sakkinen, 1988</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Fiedler, 1989</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Thielen, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Cargill, 1993</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Shopiro, 1993</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Reed, 1993</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Moreland, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Reed, 1994a</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Davis, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>Firesmith, 1993b</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Fault hypothesis: C++</td>
<td>D'Souza and LeBlanc, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Fiedler, 1989</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Cheatham and Mellinger, 1990</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Harrold et al., 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Thuy, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>McCabe et al., 1994a</td>
<td>Safe</td>
<td>C++</td>
<td>OOTool</td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Smith and Robson, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Fiedler, 1989</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Thuy, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Smith and Robson, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Kung et al., 1993 and 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Turner and Robson, 1993c</td>
<td></td>
<td>C++</td>
<td>MKTC</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Parrish et al., 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Harrold and Rothermel, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>D'Souza and LeBlanc, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>McCabe et al., 1994a</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>McGregor and Dyer, 1993a,b; McGregor, 1994a,b; McGregor and Korson, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Poston, 1994</td>
<td>OMT</td>
<td>C++</td>
<td>SiP/T</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Thuy, 1992</td>
<td>CDL</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Lieberherr and Xiao, 1992</td>
<td></td>
<td>C++</td>
<td>Demeter</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Kung et al., 1993</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Lee et al., 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Jüttner et al., 1994c</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Regression test</td>
<td>Rothermel and Harrold, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Leiter et al., 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>McCabe, 1993</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Moreland, 1994</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Lakos, 1992</td>
<td></td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Subject</td>
<td>Source</td>
<td>Method</td>
<td>Language</td>
<td>Tool</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Thuy, 1992 and 1993</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Smith and Robson, 1992</td>
<td>C++</td>
<td>FOOT</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Turner and Robson, 1992</td>
<td>C++</td>
<td>MKTC</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Jia, 1993</td>
<td>Z</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Graham et al., 1993</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Hoffman and Strooper, 1993</td>
<td>C++</td>
<td>TestGraphs</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Dorman, 1993</td>
<td>C++</td>
<td>Cantata</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Wilke, 1993</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Jia, 1993</td>
<td>Z</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Parrish et al., 1994</td>
<td>C++</td>
<td>ipt</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Oracle</td>
<td>Hoffman and Strooper, 1993a,b</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Oracle</td>
<td>Jia, 1993</td>
<td>Z</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Thilen, 1992</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Turner and Robson, 1993d</td>
<td>C++</td>
<td>MKTC</td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Davis, 1994</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Davis, 1994</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Binder, 1994d</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Coleman et al., 1994</td>
<td>Fusion</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>McCabe and Watson, 1994</td>
<td>C++</td>
<td>OOTOOL</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Martin et al., 1989</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Fiedler, 1989</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Thuy, 1992 and 1993</td>
<td>CDL</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Arnold and Fuson, 1994</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Jönnert et al., 1994a,b,c</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Testability: complexity</td>
<td>Li and Henry, 1993</td>
<td>Classic Ada</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Chau and Smith, 1994</td>
<td>CLOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: Black-box testing</td>
<td>Liskov and Guttag, 1986</td>
<td>LARCH</td>
<td>CLU</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>ADT: White-box testing</td>
<td>Liskov and Guttag, 1986</td>
<td>LARCH</td>
<td>CLU</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: Black-box testing</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Doong and Frankl, 1991</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Meyer, 1992</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Frankl, 1989</td>
<td>Eiffel</td>
<td>ASTOOT</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Doong and Frankl, 1991, 1992</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>6.2</td>
<td>Testability: uniformity</td>
<td>Meyer, 1992</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Wong, 1992</td>
<td>Eiffel</td>
<td>ACE</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Murphy and Wong, 1992</td>
<td>Eiffel</td>
<td>ACE</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Giltinan, 1994</td>
<td>Eiffel</td>
<td>TestManager</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>D'Souza and LeBlanc, 1994</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Oracle</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Giltinan, 1994</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Murphy et al., 1994</td>
<td>Eiffel</td>
<td>ACE</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Gindre and Sada, 1989</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Murphy and Wong, 1992</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Murphy et al., 1994</td>
<td>Eiffel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Capper et al., 1994</td>
<td>ENFIN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>ADT: formal verification</td>
<td>Olhoff, 1986</td>
<td>Mod-Pascal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Trausan-Matú et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Trausan-Matú et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Fault taxonomies</td>
<td>Trausan-Matú et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation-based class test</td>
<td>Trausan-Matú et al., 1991</td>
<td>mXRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Cox, 1988</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Wilke, 1993</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Cox, 1988</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Taenzer et al., 1989</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Taenzer et al., 1989</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Love, 1992</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Cox, 1988</td>
<td>Objective-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Subject</td>
<td>Source</td>
<td>Method</td>
<td>Language</td>
<td>Tool</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Firesmith, 1993a</td>
<td>ADM3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Booch, 1991</td>
<td>Booch OOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Thuy, 1992</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Experience</td>
<td>Thuy, 1992 and 1993</td>
<td>CDL</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Testability: dependency</td>
<td>Desfray, 1994</td>
<td>Class Relation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Desfray, 1994</td>
<td>Class Relation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Binder, 1994b</td>
<td>FREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Binder, 1994b</td>
<td>FREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Binder, 1994b</td>
<td>FREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Binder, 1994b</td>
<td>FREE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Coleman et al., 1994</td>
<td>Fusion</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Automated model validation: formal</td>
<td>Kemmerer, 1985</td>
<td>Ina Jo</td>
<td>Pascal</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Fault hypothesis: polymorphism</td>
<td>Leavens, 1991</td>
<td>LARCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>ADT: formal verification</td>
<td>Liskov and Guttag, 1986</td>
<td>LARCH</td>
<td>CLU</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>3.3</td>
<td>Fault hypothesis: inheritance</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>3.5</td>
<td>Fault hypothesis: state related</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>7.4</td>
<td>Oracles</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>7.5</td>
<td>Built-in test</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>4.2</td>
<td>Automated model validation: formal</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>5.3</td>
<td>Fault hypothesis: state related</td>
<td>Doong and Frankl, 1994</td>
<td>Lobas</td>
<td>Eiffel</td>
<td>ASTOOT</td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>Jacobson et al., 1992</td>
<td>OOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Representation-based class test</td>
<td>Jacobson et al., 1992</td>
<td>OOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>Jacobson et al., 1992</td>
<td>OOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Jacobson et al., 1992</td>
<td>OOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Subject</td>
<td>Source</td>
<td>Method</td>
<td>Language</td>
<td>Tool</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Jacobson <em>et al.</em>, 1992</td>
<td>OOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Dauchy <em>et al.</em>, 1993</td>
<td>PLUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Zwehen <em>et al.</em>, 1992</td>
<td>RESOLVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Selic <em>et al.</em>, 1994</td>
<td>ROOM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Adequacy and coverage</td>
<td>McCabe <em>et al.</em>, 1994a</td>
<td>Safe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>McGregor, 1994b</td>
<td>SASY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Integration test</td>
<td>McGregor and Korson, 1994</td>
<td>SASY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>McGregor and Korson, 1994</td>
<td>SASY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Automated model validation: semi-formal</td>
<td>Mellor and Shlaer, 1994a,b</td>
<td>Shlaer/Mellor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Shlaer and Mellor, 1988</td>
<td>Shlaer/Mellor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Gannon <em>et al.</em>, 1981</td>
<td>SIMPL-D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>System test</td>
<td>Graham, 1994</td>
<td>SOMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Methodology/process</td>
<td>Graham, 1994</td>
<td>SOMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>ADT: black-box testing</td>
<td>Jia, 1993</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>ADT: white-box testing</td>
<td>Hayes, 1986</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Auto test case generation/execution</td>
<td>Jia, 1993</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Test drivers</td>
<td>Jia, 1993</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Oracles</td>
<td>Jia, 1993</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>