# A TAXONOMICAL REVIEW OF OBJECT-ORIENTED SIMULATION MODEL VERIFICATION AND VALIDATION TECHNIQUES

Levent Yilmaz

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
e-mail: yilmaz@vt.edu

**KEYWORDS**

Model credibility, model testing, object-oriented simulation, validation, verification.

**ABSTRACT**

The goal of this paper is to provide a taxonomical review of verification and validation (V&V) techniques for assessing the accuracy of object-oriented simulation models and to present guidelines for the selection of the techniques based on their effectiveness and costs. The techniques are evaluated based on the identified indicators that relate the techniques to their error detecting capability and the costs that measure the ease-of-use aspects. The paper introduces a preliminary object-oriented model fault taxonomy based on a comprehensive literature survey and authors' experience. The fault taxonomy not only compares and contrasts the error detection capabilities of the techniques but also provides a basis and guidance for detecting specific faults likely to occur due to complexities of object-oriented paradigm such as object/class hierarchies, state-dependent behavior, dynamic and diverse patterns of interactions among the objects. The guidelines and the comparative analysis of the object-oriented V&V techniques allow practitioners to select the techniques based on the benefits and costs.

## 1. INTRODUCTION

Object-oriented paradigm (OOP) provides many advantages such as modularity, information hiding, extensibility, reusability, and maintainability. However, object-oriented simulation models are increasingly becoming larger and complex. The various diverse patterns of interactions, inheritance, encapsulation, and polymorphism make understanding, controllability, and observability of object-oriented models difficult. There has been vast amount of research efforts on developing methodologies and techniques for analysis, design, and implementation of object-oriented systems in the past. There are several analysis and design methodologies proposed in the literature, that provide excellent background on object-oriented development [Coad and Yourdon 1991; Rumbaugh *et. al* 1991; Booch 1994; Fishwick 1995; Hill 1996; Jacobson 1992; Martin and Odell 1992; Shlaer and Mellor 1988; Wirfs-Brock et. al 1990].

Despite the advantages of OOP, practitioners and researchers face new technical challenges in the verification and validation (V&V) process. In the last few years, there has been an increasing attention in the software engineering community for the V&V of object-oriented software. Several new techniques, methodologies and adaptive techniques have been published in the literature. However, as Binder [1995] points out, only the easy problems have been reported and the most important technical challenges still remain to be addressed. The situation is even more serious for object-oriented simulation model V&V since simulation requirements are much more demanding than a typical software system.

The OOP has five main concepts: objects, classes, polymorphism, inheritance, and dynamic binding.

*Objects* are the building blocks of a model and represent the elements of interest in the modeled system. *Classes* specify and define a set of objects by providing a template and a construction mechanism to implement user-defined types. Each object is an instance of the corresponding class which defines it. *Inheritance* is an important contribution of the OOP [Korson and McGregor 1990]. Inheritance is a specialization/generalization relationship which organizes classes to facilitate sharing of attributes and common behavior. A class can be defined broadly and then refined into subclasses where each subclass inherits all of the properties of its superclass and adds its own unique properties [Rumbaugh *et al* 1991]. This relationship facilitates model reuse and extensibility [Korson and McGregor 1990].

*Polymorphism* enables an object to assume more than one form. The same operation may behave differently when defined under different classes. *Dynamic binding* means that the code associated with a given message call in the source program is not known at compile time, and is dispatched at run time according to the object that receives the call.

## 2. PROCEDURAL VERSUS OBJECT-ORIENTED PARADIGM

Procedural paradigm is based and organized around the notion of algorithmically specified functions and uses the functional decomposition strategy. In functional decomposition, the problem space is transformed into solution space in terms of interdependent functions. As a result, the system is seen as a set of procedures that operate on data. Initially the design is represented by a single function and then by refinement, the main function is decomposed into subfunctions; this decomposition continues until the desired functionality is achieved. The units (i.e., functions) in this strategy implement a single functional requirement or abstraction. The interface between the units is usually achieved via function call, global data, or parameter passing. Thus the two main entities in procedural paradigm are functions and global data. These two entities create two hierarchies: the calling structure and data structure. Figure 1 demonstrates the structure of a conventional procedural model.

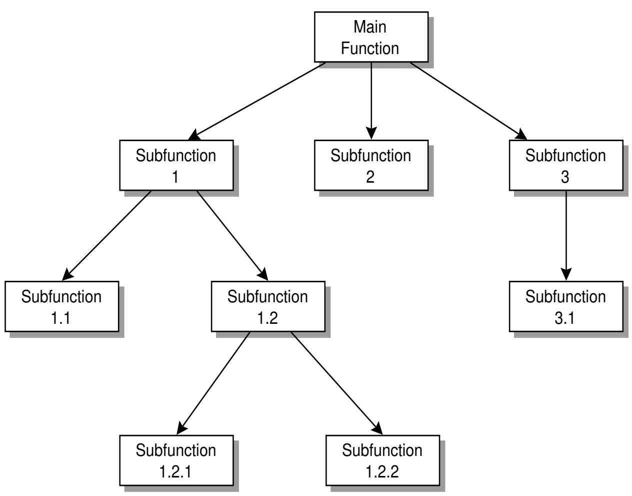


**Figure 1** Procedural Model Structure

Object-oriented model is organized around the objects which encompass state, behavior, and identity as discussed in the previous section. The objects are grouped into class hierarchies to share common behavior and structure. Objects encapsulate the state and behavior descriptions and interact with other objects by message passing.

Scenarios involving the interaction of communicating objects are used in the integration and system level V&V stages. Figure 2 illustrates a simple view of the structure of an object-oriented model.

Differences between procedural and object-oriented models affect the V&V process. The differences given in Table 1 outline some of the properties between procedural

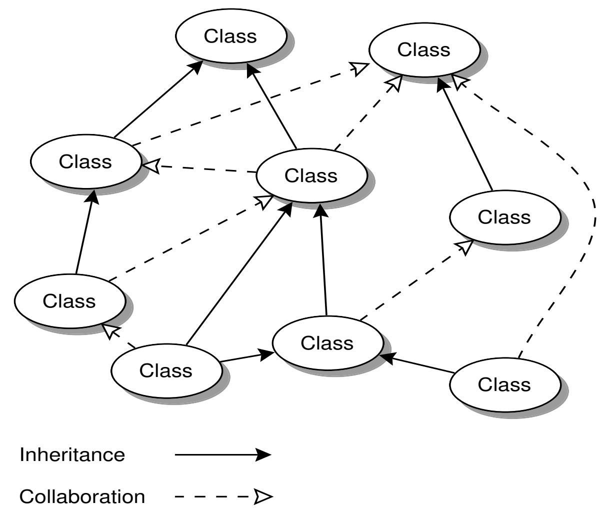


**Figure 2** Object-Oriented Model Structure

and object-oriented models and their implication for V&V. Some of the advantages of object-oriented paradigm are modularity, information hiding, reusability, extensibility, and maintainability. However, this technology does not make V&V unnecessary. In fact, the increased modeling power of OOP introduces new sources of errors and new complexities to the V&V process.

**Table 1.** Procedural vs. Object-Oriented Model

| **Procedural Model** | **Object-Oriented Model** |
|---|---|
| Unit Testing = Procedure Testing | Unit Testing = Class Testing |
| White-Box Testing Emphasized | Black-Box Testing Emphasized |
| Functional Decomposition | Top-Down, Bottom-Up (Middle-Out) OOAD |
| Waterfall Lifecycle | Incremental Iterative Lifecycle |
| Subroutine Calls | Message Passing |
| Importance of Structure in Integration | Importance of Behavior in Integration |
| Imperative Nature | Declarative, Event-Driven Nature |

Objects exhibit state dependent behavior. That is, the output of an object in response to a message call depends on the messages it has received before. Since the effect of an operation on an object depends on the current state and may change the state of the object, combined effects of the operations must be verified. However, the problem of state testing is difficult due to the large number of states in the specification of an object-oriented simulation model.

Another problem arises because of the diverse patterns of interactions among objects at runtime. The interactions causes delocalization of the plans by delegating or passing the computation to the objects which are responsible for carrying out the task. While the runtime behavior of a well structured conventional program is intellectually tractable and easy to understand, controlling and observing the object-oriented behavior becomes very complex due to these diverse interactions and fragmentation of plans [Binder 1995]. Moreover, inheritance and polymorphism introduce undecidability to program-based testing [Barbey and Strohmeier 1994]. That is, the dynamic binding feature introduces paths and states at runtime which are not obvious from the source code. This reduces the effectiveness of the conventional informal techniques such as walkthroughs, reviews, and inspections. Binder [1994] also agrees that the fragmentation of functionality due to inheritance, dynamic binding and cooperative control strategies which result from distribution of control flow and state control over several classes decrease comprehension and make static verification techniques less useful [Linos and Courtois 1994; McCabe and Watson 1994]. Moreover, the testability of the model becomes difficult since controllability and observability is impaired [Binder 1994b; Voas 1996]. Kung *et al.* [1995] summarize these problems under two categories: the understanding problem and the complex dependency problem. The understanding problem arises due to the encapsulation and information hiding principles of OOP. These features result in long chains of method invocations and reduces the comprehensibility of the resulting model structure. The complex dependency problem is caused by the relationships in the object-oriented program. Three main relationships exist: inheritance, aggregation, and association. The dependencies resulting from these relationships cause difficulties in understanding a given class, identifying a starting point for testing in a complex object-oriented model, constructing test stubs, and identifying and verifying the effect of dynamic binding and polymorphism.

Another study that describes some of the problems that might be encountered during the testing of object-oriented systems is given by Smith and Robson [1990] and Cheatham and Mellinger [1990]. The authors discuss validation problems during both development and maintenance. They disagree with the industry's opinion on seeing object-orientation as the solution to development problems. They state that inheritance increases the V&V required. In case where all the features of the base class are inherited by the subclass, it is easiest to test since the test cases for the parent class can be used for the derived class. However, extra test cases are necessary for the overrided and redefined methods. Then the authors discuss the problems with multiple and repeated inheritance, genericity, and polymorphism. One of the interesting problems identified is the validity of the use of inheritance itself. However, the validity of the use of inheritance is a subjective decision and can be tested manually by domain dependent V&V techniques.

The nature of object-oriented simulation model development is incremental and iterative. This requires a more iterative V&V process which is usually stated by the following phrase: "analyze a little", "design a little", "code a little", and "test a little" [Booch 1991]. However, the iterative nature of development and hurry-up management produce code lacking up-to-date specifications and documentation which affect the V&V process [Binder 1995].

The basic principles and concepts of OOP themselves introduce some implications for V&V. Binder [1994] argues about the implications of encapsulation, inheritance, polymorphism, and several other concepts related to object-oriented programs. The basic problems he identifies are the amount of retesting in the case of inherited features, verifying the state dependent behavior in the case of distributed control and state among a set of objects, and the development of an effective V&V strategy given the incremental and iterative nature of object-oriented development. Although encapsulation itself is not a direct problem for V&V, it becomes an obstacle to the observability of the class under testing.

## 3. TAXONOMICAL REVIEW OF OBJECT-ORIENTED VV&T TECHNIQUES

This section presents a taxonomy of V&V techniques, brought from conventional and object-oriented software testing literature, that are applicable for object-oriented simulation models.

Figure 3 presents the taxonomy of V&V techniques in three approaches that are characterized by the degree and manner they incorporate other paradigms.

1. *The conventional approach:* In this approach, existing well-known techniques are used for V&V of object-oriented simulation models without any adaptation or extension for OOP. In particular, these techniques are applicable for method level testing since the methods of classes are actually functions and procedures that use imperative language constructs. The conventional techniques are given in Figure 4.[Balci et al. 1996]
2. *The adaptive approach:* There has been a lot of V&V research conducted for the procedural paradigm. Some

**Figure 3** A Taxonomy of V&V Techniques for Object-Oriented Simulation Models

procedural V&V techniques can be used in a new way by adapting or extending to include object-orientation. Figure 3 shows the adaptive techniques that can be used for object-oriented simulation model V&V.

3. *The specific approach:* This approach contains new techniques created based on object-oriented formalisms and are intended for the sole purpose of object-oriented software V&V. Figure 3 shows the V&V techniques that are specifically applicable for object-oriented simulation model accuracy assessment.

### 3.1 Adaptive VV&T Techniques

#### *3.1.1 Adequacy Criteria-Based Testing*

Perry and Kaiser [1990] discuss the theoretical view of object-oriented testing by describing the applicability of Elaine Weyuker's test adequacy criteria. Although the criteria is originally designed for conventional systems, it is useful enough to determine adequate level of testing for object-oriented models. The authors list each of the criteria and discuss the applicability to object-oriented paradigm. Detailed examples are provided to clarify the effect of each criterion. Only four of the eleven criteria are found to be applicable to object-orientation. The criteria are as follows:

*Antiextensionality:* if two programs compute the same function, a test set adequate for one of them is not necessarily adequate for the other one. This criterion states that functional testing must not be the only testing method, but we must use structural testing techniques to develop test cases that depend on the implementation of the functions.

*General Multiple Change:* if two programs are syntactically similar, a test set adequate for one program is not necessarily adequate for the other. This implies that functional testing is not the only testing method performed.

*Antidecomposition:* if the surrounding environment of a unit changes, it will require retesting.

*Anticomposition:* Unit testing in isolation is not adequate; interactions occur during composition and new sources of errors become visible during integration.

The above criteria provide several important implications for testing. The antidecomposition principle requires the retesting of all inherited features. The antiextensionality criterion covers the case for overriding methods. That is, new test cases must be developed since the test cases for the older version will not be adequate for the new version. Intuitively, if the implementation of a class changes, but its functionality is the same, it would be expected to retest only the modified class. On contrary, the anticomposition criterion states that all other classes that use the modified one must be retested.

### *3.1.2 Class Testing by Pointer Examination*

Another interesting approach to testing is to adapt aliasing analysis techniques to uncover errors. D'Souza and LeBlanc [1994] examine the use of pointer aliasing as a testing technique. They summarize and evaluate the existing techniques and then show that using pointer aliasing information can be as powerful in its ability to detect errors as other techniques. They argue against the use of state-based testing since objects can not be treated as finite state machines due to large state space and the parameters passed to the methods. The basic idea is to traverse the run time structure obtained by the test case execution. This structure is stored in a table of pathnames. Each entry of the table is composed of a pathname, an object identifier, and the dynamic type of the object associated with that pathname. Then the table is sorted on object identifiers. Finally, the table is scanned for duplicate object identifiers, which appear adjacent due to sorting. The duplicates indicate that the corresponding pathnames alias the same object, which can be an anomaly.

### *3.1.3 Adapted Data Flow Testing*

One of the well known testing technique in the context of procedural paradigm is data flow testing. Although conventional data flow testing has been successfully applied to testing of programs developed using the procedural paradigm, it is not powerful enough to be used for object-oriented paradigm. However, there are new adapted data flow techniques that are appropriate for object-oriented paradigm.

#### 3.1.3.1 Flow Graph-Based Class Testing

Parrish et al. [1993] also argue against the use of only specification-based techniques. They propose a theory for the adaptation of conventional flow graph-based strategies to testing object-oriented programs. They develop a formal framework to model classes with flow graphs. Then analogous definitions of node coverage, branch coverage, path coverage, definition coverage, use coverage, and du-path coverage are defined on the class flow graph. Each node in the graph refers to a method of the class. The edges between nodes represent the feasible sequence of method invocations which can be derived from the model-based specification of the class.

#### 3.1.3.2 Hierarchical Data Flow Testing

Hierarchical data flow testing technique, which is an extension of the existing data flow techniques, is proposed in [Subramanian et al. 1994]. The authors decompose an object-oriented program into three different hierarchies: classes, objects, and attributes. They describe class flows, object flows, attribute flows and then define the interrelationships among them. The definitions such as defined, referenced, and killed which are used in traditional data flow testing are extended for class, object and attribute flows. The analysis is based on an algorithm which has two phases. In the first phase class flow, object flow, and attribute flow is performed. In the second phase the ambiguities raised during the class and object flow stages are resolved. Similar to the previous approaches, the authors show the usefulness of the method for method sequence anomalies. One of the most important contributions of this approach is that it shifts the focus from variables to objects and classes; thus anomalies at this level of abstraction can also be revealed.

#### 3.1.3.3 Intra-Class Data Flow Testing

Another technique for adapting data flow testing to object-oriented paradigm is discussed in [Harrold and Rothermel 1994]. They argue that most of the existing approaches are based on specification-based testing techniques and there is a necessity to augment these approaches with code-based or white box techniques. Their technique defines three levels to perform data flow class testing:

- *intra-method testing:* performs dataflow testing for individual modules,
- *inter-method testing:* tests the interaction of methods inside a class,
- *intra-class testing:* tests the sequence of calls that can be initiated by the user of the class.

**Figure 4** Conventional V&V Techniques

Each class is represented as a single entry, single exit program and then a class control flow graph (CCFG) is constructed. CCFG shows the calling sequence of the methods inside the class. This control flow graph is then used to perform interprocedural data flow testing, which is adapted from an interprocedural data flow analysis algorithm presented in [Pande et al. 1994]. Their technique is also useful in determining which sequences of methods should be executed to test classes even in the absence of specifications. However, the sequence derived from the data flow analysis is consistent in the context of the system under test. If the class is used in a different system as a reusable component then its usage and so its method sequence may be different.

### 3.1.4 *Domain Dependent Testing*

Most of the approaches suggested for object-oriented testing do not mention the domain structure and rules for which the model is developed. However, the structure of the domain and the knowledge structure represented by the relationships among objects also need verification. Kandelin and O'Leary [1995] extend their previous research on rule-based system verification to develop verification approaches for object-oriented AI models.

They use meta-knowledge from the domain to examine the completeness, consistency, correctness and redundancy of the object model. Domain independent verification approaches concentrate on the nature and structure of the objects. The authors provide several guidelines when followed helps finding ambiguities, inconsistencies, and redundancies in the class and instance (object) structure. They also suggest the use of inspection objects that perform verification tasks. However, these inspection objects are required to have sufficient knowledge about the domain of the particular system.

### 3.1.5 *Extended Data Flow Analysis*

In [Chen and Low 1995] an extension to traditional data flow analysis is proposed. The technique uses the memory location of variables and class members rather than using improper sequence of actions on a variable. This allows the detection of anomalies based on pointer variables and class operators. Moreover, to detect anomalies it is not enough to check only the current state value and the type of action (i.e., define, reference, undefined), but also the class from which the action is requested. Thus with the incorporation of class information, an extended state transition diagram and new action types are introduced. They finally discuss the feasibility of applying these ideas to cover objects and their member function application sequences. The approach is to capture the semantics of the class by imposing a method invocation sequence for the class under test. However, a more detailed analysis of this idea is given in [Kirani and Tsai 1994] which provides both a method sequence specification and a test technique based on this specification.

### 3.1.6 *Fault-Directed Testing*

By identifying faults specific to object-oriented paradigm, one can develop testing strategies either by adapting conventional techniques or by developing pure object-oriented approaches. Hayes [1994] focuses on the necessity of developing fault taxonomy to guide testing. She examines several fault types and associated verification techniques that are mentioned in [Firesmith 1993] and [Purchase and Winder 1991]. She argues that in order to adapt conventional techniques it is necessary to understand the specific components of an object-oriented system. She identifies five components: (1) objects, (2) associated messages and methods, (3) class hierarchies (inheritance, aggregation, association relationships), (4) external interfaces, which are connections to databases, users, communication channels etc., and (5) tools and utilities. The tools and utilities are regarded as highly usable components. She argues that these components require a certification procedure since they will be reused in a variety of systems without customization. The methodology developed by Wohlin and Runeson [1994] provides a useful technique, based on usage modeling, for certifying components of a model.

Firesmith [1993] analyzes the problem of testing and recommends a hierarchical testing strategy based on the existing techniques. He identifies both the differences between the object-oriented and procedural paradigm and their implications for testing. Based on the levels of testing (i.e., object, class, subassembly, message, exception, scenario) a taxonomy of errors and their priority for testing are presented. Then he suggests a verification technique for each error type listed in the taxonomy. The techniques proposed are conventional black-box testing, white-box testing and inspections to detect these specific errors. However, he does not explain how to apply the techniques to uncover errors.

### 3.1.7 *Hypothesis-Based Testing*

This approach, based on the specification-based testing, is described in [Barbey 1995]. The author adapts the theory of specification-based testing developed by Bernot et al. [1991] to object-oriented paradigm. The theory is composed of two main sections: a theory of test selection and a theory of test satisfaction. In particular, the aim is to use hypothesis to reduce the size of a test set such that the resulting test set becomes valid, finite, decidable and complete. The author adapts the regularity and uniformity hypothesis to object-oriented paradigm and then adds incrementability hypothesis for reusing the test sets of the parent class for the derived class.

The regularity hypothesis helps select and reduce test sets by identifying the equivalence classes of states which are behaviorally equivalent.

The uniformity hypothesis is defined as follows: if a program behaves correctly for a property A which is an instantiation of type S, then every property which is an instantiation of A and related to type S must behave correctly [Barbey 1995]. The author analyzes this hypothesis by considering the subtype relationship in object-oriented software. The incrementability hypothesis is proposed for reducing the test sets in the presence of inheritance by reusing the transitions that must be traversed when testing a derived type.

#### *3.1.8 Modal-Based Testing*

Class modality is one way of defining the patterns of behavior of objects. Binder [1996b] identifies four different modalities for classes: nonmodal, unimodal, quasimodal, and modal. Each modal category has a different behavior pattern based on the message sequence and domain constraints on the classes.

A *nonmodal class* imposes no constraints on the sequence of messages. For nonmodal classes Binder applies the *vertex probe strategy*, which is an adaptation of domain and loop testing techniques discussed in [Adrion et. al 1982; Beizer 1990; Lee 1987].

A *unimodal class* restricts the sequence of messages that it can expect at runtime. The testing strategy for this type of classes is to apply the expected and unexpected sequences to the object under test and to check for conformance of the resultant state with the specified state. The sequences can be identified by tracing the transitions in the state model of the class.

A *quasimodal class* restricts the sequence of messages based on its domain constraints. For instance, a stack object allows any order of push and pop operations, unless it is full or empty. Quasimodal classes can be tested by characterizing the states by constraints (i.e., instance variable domains) and then using vertex probe strategy to the parameters used as constraints.

A *modal class* restricts the message sequences based on both the domain and sequence constraints. Thus the testing strategy follows a hybrid approach; that is both vertex probe strategy and message sequence captured in state model are used in uncovering errors.

#### *3.1.9 Three-View Model Testing*

This approach focuses on test case generation and testing using different views of the system: object model, dynamic model, and functional model. Usually these techniques are influenced by the Object Modeling Technique (OMT) discussed in [Rumbaugh et al. 1991] for the design and modeling of object-oriented systems. Thus the test cases are prepared based on three different models: (1)Object Model, (2) Functional Model, (3) Dynamic Model.

##### 3.1.9.1 Test Case Generation from Object Models

Poston [1994] discusses a project in which automated testing tools are integrated with the OMT life cycle. The object model identifies the classes, attributes and their relationships (i.e., inheritance, association, aggregation). Although object model incorporates name, type, and initial values of attributes, it does not include domain descriptions. The testers add textual annotations to the object model to make it test-ready. The dynamic model introduces graphical notations for the specification of events, states, transitions, and hierarchies of states. However, creating a test-ready dynamic model is hard and time consuming. After creating test-ready models, a test case generator is used to develop test cases based on the specification-based test design techniques such as boundary value analysis, equivalence class testing, cause-effect graphing, event-based testing, and state-based testing. Hence although test cases are developed from object-oriented design models, the techniques in extracting them from the models are still traditional black-box strategies.

##### 3.1.9.2 A Combinative Test Model

Kung et al. [1995b] developed a methodology that uses a test model composed of three views of the system. Those views are represented by three types of diagrams: 1) the object relation diagram (ORD), 2) the block branch diagram (BBD), and 3) the object state diagram (OSD). Those diagrams are extracted from the source code using reverse engineering technique [Kung et al. 1993].

The ORD diagram represents the aggregation, association, and inheritance relationships. The BBD deals with those aspects of the system concerned with the control structure, transformation of values, and functional dependencies. The OSD is the dynamic model of the model; it represents the states and sequence of transitions graphically with state diagrams. The test model is used in generating test cases, creating a testing order, and providing a class firewall for regression testing. The test cases are developed using each diagram. The ORD is used to identify disconnected directed subgraphs which can be tested in parallel. In case directed cycles do not exist topological sorting is used to develop a testing order. The BBD is used as a data flow diagram and traditional structural and functional testing techniques are applied to develop test cases.

The OSD diagram is used for object state testing. For each instance variable an Atomic Object State Diagram (AOSD) is constructed. The states are identified in terms of the disjoint intervals of instance variable's domain. Then a Composite Object State Diagram (COSD) is constructed by combining the Sods. The process of generating test cases from the COSD is similar to Chow's [1978] technique. A test tree whose nodes are composite states of COSD is constructed. The edges of the tree represent the transitions between the composite states. If the COSD contains n AOSDs, then each state is identified by an n-tuple where the i-th component of the tuple represents the state of the i-th AOSD. The details of the algorithm for construction a COSD, and generating test cases from the composite diagram are described in [Kung et al. 1994]. A test model based on these three views have both advantages and disadvantages. The usefulness of the test model is summarized in [Kung et al. 1995b] as follows:

The tester and maintainer can understand the structure and relationships among the components of an object-oriented program.

It allows the tester and maintainer to find strategies based on the test model to reduce testing effort. It facilitates the definition and analysis of test criteria.

#### 3.1.9.3 System Specification Model Testing

This approach is advocated in Software Architects' Synthesis model (SASY) by McGregor and Korson [1994] and it is part of a comprehensive testing process utilizing the iterative approach. They represent SASY as a tool to structure the development process and to allow the integration of testing process during each iteration stage.

Three main categories of testing are identified: testing the development process, testing analysis and design models, and testing implementation. They focus and suggest strategies for the last two categories, namely, testing the implementation and models developed during the analysis and design phases. The system model testing process is based on the completeness, consistency, and correctness criteria. Use-cases are identified to develop interaction models, which are checked for errors, ambiguities and omissions.

The Class-Responsibility-Collaboration (CRC) cards are prepared for modeling individual classes during the specification phase. Then several checks are performed on these CRC cards to demonstrate their correctness and consistency. The testing proceeds by using the models of OMT technique; that is, object model, dynamic model, and functional model. During this phase not only the accuracy of the transformation from the analysis model to design model must be verified but also the consistency of these different views must be considered. For the system model testing process the authors propose guidelines for testing the three different views of the system: object model, functional model, and dynamic model.

However, the use of models in an ad hoc manner is one of the major reasons behind errors detected by validation. The problem is compounded when more than one modeling technique is used to represent the model specification and design [Lalioti and Loucopoulos 1993]. Furthermore, browsing between different models does not help developer. Instead the developer needs a unified understanding of the models. That is, the interaction among the models must be captured to check also for the inconsistencies and ambiguities.

## 3.2 Specific VV&T Techniques

The testing process is usually defined as the generation of the test cases which has the input and expected output pairs, such that the program processes the input and then compares the actual output to the expected output (i.e., oracle) to decide whether test case is successful or not. Given the differences between object-oriented and procedural model, it is necessary to develop object-oriented approaches to testing. The object-oriented processing model is one in which the data is active and the processor is passive whereas the processing model of procedural model, which is based on Von Neumann model of processing, is based on active processor and passive data. This modeling difference is argued in [Smith and Robson 1990] to demonstrate that the testing process defined above is not suitable for object-oriented paradigm.

### *3.2.1 Algebraic-Specification-Based Testing*

These are the approaches using algebraic specifications to object-oriented testing.

#### 3.2.1.1 DAISTS Testing Strategy

One of the earliest systems to address the question of testing data abstractions is DAISTS (Data Abstraction Implementation Specification and Test System) [Gannon et al. 1981]. In DAISTS the axioms of an algebraic specification are used as an oracle for the testing implementations of the Abstract Data Type (ADT). A test case consists of a set of arguments for the left-hand side of the axiom. The test case is given as input to the left-hand side and right-hand side of an axiom and then the results are checked by a user-supplied equality (EQN) function.

A similar approach is proposed by Daniel Hoffman [Hoffman 1989]. His approach is based on traces, which are sequences of procedure or function calls, with parameter values. A test case is represented as a five-tuple: <trace, expexc, actval, expval, type>. Expexc is the exception that may be generated by the trace during its execution. Actval refers to the actual value obtained after the execution of the trace. Expletive is the expected value serving as an oracle. Finally, the type is the type of the result returned by the application of the trace.

#### 3.2.1.2 ASTOOT Testing Strategy

Doong and Frankl [1994] describe an approach, which is called ASTOOT (A Set of Tools for Object-Oriented Testing). It is based on the algebraic specification of ADTs. Test cases consists of pairs of method invocation sequences along with a tag. The tag reflects whether two sequences should produce the same result in the state of the ADT. A class is referred as the implementation of the corresponding ADT. A class is considered to be a correct implementation of the ADT if and only if, for every pair of method invocation sequences $s_1$ and $s_2$, applied to a pair of objects o1 and o2, the objects are observationally put into equivalent states [Turner and Robson 1993a].

ASTOOT consists of three components: the driver generator, compiler, and simplifier. The user is allowed to generate a sequence of operations from algebraic specification using an interactive test generation tool. The test generation tool is based on the compiler and the simplifier. The simplifier uses rewrite rules provided by the compiler to produce two equivalent sequences of operations which form a test case. Then the test cases are executed by the driver generator.

ASTOOT approach is a specification-based testing technique; it only tests the functionality of the class without considering the internal structure and implementation details. The ability of the approach to accept manual (hand-crafted) scripts for situations when formal specifications are not available is seen as an advantage over other techniques. Moreover, rather than using functional notations of algebraic specifications, ASTOOT uses the specification technique called LOBAS which is similar to a trace specification.

#### 3.2.1.3 DAISTISH Testing Strategy

A more recent approach, called DAISTISH, is proposed by Hughes and Stotts [1996]. DAISTISH is a perl script which uses the formal specification of an ADT; it combines this specification with the actual code and then creates a test driver. DAISTISH is similar to DAISTS; however, DAISTISH is more effective in creating test drivers for languages which use side effects to implement ADT.

This approach is also similar to ASTOOT; however the authors concentrate on the semantic specific issues of correctly duplicating the objects, rather than automating the test case selection problem. The differences between these tools are summarized in [Hughes and Stotts 1996].

### *3.2.2 Class Firewall Technique*

The retesting and class firewall concepts developed by Kung et al. [1995a, b] worth mentioning. The programs undergo continual changes. The procedural paradigm provides well-defined tractable structure by the use of functional decomposition principle. Hence the effects of a change can be traced easily. However, in the case of object-oriented model, the effects must be propagated to all the objects which have relationships to the object under change. The Object Relation Diagram (ORD) and Block Branch Diagram (BBD) is used for identifying the regions of the code which needs retesting. The details of construction of the class firewall, its use in retesting, and an algorithm for creating an optimal test order for the firewall is described in [Kung et al. 1995a].

### *3.2.3 Component Certification Technique*

Reuse is one of the key issues in object-oriented development. Classes, components, and highly reusable model libraries can be reused for different problem domains without wasting the effort for redeveloping the same functionality. However, since these components will be reused in a variety of systems without customization, they require a certification procedure. The methodology developed by Wohlin and Runeson [1994] provides a useful technique, based on usage modeling, for certifying components of a system. The objective of their method is to obtain reliability measure for the components. During development for reuse, they suggest that components can be certified with usage testing by developing usage profiles and then applying a certification model that provides the level of confidence in them. They generate test cases by using a hierarchical usage model by a state hierarchy model which uses Markov chains to model the state transitions. They attach probabilities to usage and behavior transitions and use these measures to develop test sequences. The test sequences facilitates the collection of failure data and provide a basis for the certification measure.

### *3.2.4 Flattened Regular Expression Testing Strategy (FREE Approach)*

The flattened regular expression testing strategy is based on the state machine representations of the classes and clusters in the software. The test cases are selected such that the state space of the class and clusters are covered [Binder 1994a]. The flattening refers to the inclusion of the inherited features in the state model of the class under test to facilitate adequate testing of inherited features. Test cases are sequences of transitions that cover the state machine. The class state-space is constructed from the class attribute domain spaces; that is the boundary values (i.e., minimum and maximum) of the attributes identify the states of the class. The author also mentions the FREE-flowgraphs that represents the intra-class and inter-method flows, where intra-class flows are the sequence of public method calls and inter-method flow is the control flow between two methods of the class. Thus, it facilitates the identification and instrumentation of data-flow paths among all class methods. In [Binder 1996a] the author discusses the FREE approach for system testing. At the system level the user-centered approach is investigated and use-cases are employed to develop a system test plan. The strategy can be applied at three levels:

- Functional Compliance: This level requires use-case and event-trace coverage. A formal test plan is prepared for all use-cases identified.
- Reliability Optimization: This level requires functional compliance testing to be conducted along with the operational profile to optimize reliability for a given testing budget.
- Integrity Verification: The goal of level three is to establish a system test completeness by the demonstration of thread coverage of level two test set.

The FREE approach to system testing starts with OOA/D specifications, use-cases, and interaction diagrams. Then object models are made test ready by the operational relations which are test cases obtained from class and system attributes.

### *3.2.5 The FOOT (Framework for Object-Oriented Testing) Testing Strategy*

In [Smith and Robson 1992] the authors describe the problems of testing and develop an experimental framework, named FOOT, to provide a repeatable and object-oriented approach to testing. They also argue that since the object-oriented programs are not executed sequentially and the application order of methods can be arbitrary, the testing process becomes a searching problem. Based on their observations of problems and the searching argument, the authors develop strategies for testing. The strategies incorporate several features: minimal, exhaustive, tester guided, inheritance, memory, data flow, identity, and set and examine. The three techniques, namely identity, set and examine, and inheritance can be considered as object-oriented specific testing methods:

#### 3.2.5.1 Identity Method

This testing strategy searches for pairs of sequences of methods that leave the state of the object as it was initially. The tester uses these sequences to ensure that the resultant state is the desired unaltered state.

#### 3.2.5.2 Set and Examine Method

In this method pairs of methods are designed to set and examine a particular state of the object. These methods are used in conjunction with a test driver.

#### 3.2.5.3 Inheritance Method

This method uses regression analysis to determine which parts in the inheritance hierarchy of the system need retesting after a change is made. This applies to subclasses inherited from the parent class. However, the authors do not give a detailed algorithm that identifies the regions to be retested. An incremental algorithm for testing inheritance structures is given in [Harrold et al. 1992] and will be discussed later in this section.

### *3.2.6 Inheritance Testing*

#### 3.2.6.1 Hierarchical Incremental Testing

Harrold et al. [1992] discuss an incremental approach for the testing of class hierarchies. They suggest an order for testing the inheritance hierarchy starting from the top. That is, testing starts with the base classes, and continues down the hierarchy. The testing process progresses by testing features in isolation and then testing the interactions among features. Their algorithm uses test histories from the parent classes. The test history of a parent class is incrementally updated with the information about the derived class's differences from the parent class. The new test history allows the tester to identify the reusable test cases from the parent class along with any attributes for which new test cases must be generated. Their approach supports three levels of attribute visibility: hidden, accessible to derived classes, and accessible to all classes. They model the inheritance concept in terms of a class and a modifier, resulting a derived class.

#### 3.2.6.2 Repeated Inheritance Testing

Another study related to inheritance is given by [Chung and Lee 1992]. The authors show that when repeated inheritance occurs errors may arise due to name-confliction.Figure 5 illustrates the repeated inheritance situation among classes A, B, C, and D. The authors build a directed graph of the inheritance structure where nodes denote the classes and edges denote the inheritance edges. The root classes are traversed using breadth-first traversal and then parent classes are added to the ancestor sets of the child classes. The ancestor sets are examined to find the repeated inheritances. The repeated inheritance structures are classified in a hierarchy based on their euler region numbers (r). Then these hierarchies are used in testing as follows:

> URI(1): every class in a repeated inheritance must be tested at least once.
>
> URI(2): every repeated inheritance with r=2 must be exercised.
>
> URI(3): every closed region with r=3 must be exercised.

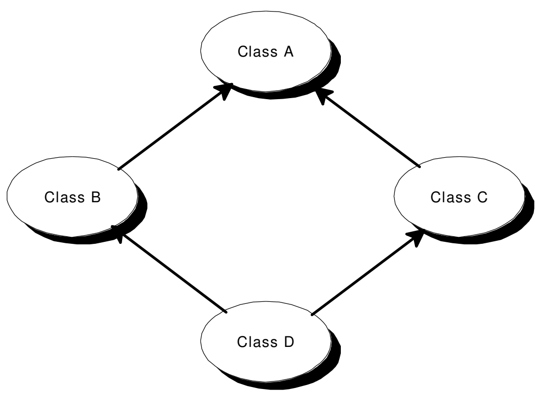


**Figure 5** Repeated Inheritance

The authors propose using McCabe's cyclomatic testing strategy after finding the hierarchies.

### *3.2.7 Integration Testing*

#### 3.2.7.1 Composition-Based Integration Testing

Structural aspects of the procedural models is helpful in testing due to the functional decomposition principle and the well-defined tractable model structure. Testers use this decomposition structure during integration testing in a top-down or bottom-up manner. However, the nature of object-oriented paradigm is event-driven. For simple programs it is easy to obtain behavior from the structure, but as the software becomes complex our ability to comprehend the software from the structure degrades.

Jorgensen and Erickson [1994] discuss this problem of structure and behavior in the case of object-oriented software. They argue that the event-driven nature of object-oriented software forces a declarative spirit on testing. Then they suggest five different levels of testing and two constructs which would be helpful during integration testing. The testing levels postulated are: method, message quiescence, event quiescence, thread testing, and thread interaction testing. Their emphasis is particularly concentrating on the behavior of software rather than structural properties. This is due to the fact that software testing concerns with behavior not structure.

They consider method testing as the unit level testing. However, this conflicts with the well-accepted notion of class as the unit level testing construct. Class testing includes both method and intra-method (interactions of the methods of a class) testing. The two constructs proposed for the integration level of testing are as follows:

Method/Message Path (MM-Path) is a sequence of member function invocations which starts with a method and ends when a method which does not issue any messages is reached.

Atomic System Function (ASF) is an input event followed by a MM-Path and ends with an output port event.

Those constructs provide a framework for integration testing. However, as the systems become larger and complex it is necessary to formally specify and capture the object interactions through MM-Paths.

#### 3.2.7.2 Incremental Integration Testing

Overbeck (1994) provides a detailed integration level testing strategy for object-oriented systems. He focuses on developing a testing infrastructure based on the relationships between the classes with different inheritance schemes. In this approach each class under test passes the self test (class unit test) if for all methods in the class message input/output is correct, all intra-class method sequence conforms to the expected sequence, and every server class of the class under test is used appropriately. Once the class passes the self test, the strategy focuses on the client/server relationships (collaborations and interactions) among the classes (i.e., contract-test). Considering two classes A and B where A uses a service provided by a method of B, the class A passes the test if it passes the self test (class unit test), it meets the preconditions of B, and A uses B's output correctly. The strategy is built on these two test suits: self-test and contract-test). The author considers different cases depending on the inheritance variations. (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.

Incremental integration testing approach uses stubs to break the cycle when the contract-test can not proceed due to the cyclic usage relationship between classes. This increases the cost of testing, but this incremental strategy aims to minimize the number of stubs and the reduce the test cases.

### 3.2.7.3 Interaction Testing

This technique is advocated by McGregor and Korson [1994] in their testing process model. They identify two levels of interactions. Interactions among the methods of a class and interactions among the methods of different classes due to the association relationships between them. They suggest investigation of pair of methods that directly or indirectly effect each other to produce incorrect results. The strategy proceeds by constructing test cases that identify attributes that are modified or accessed by two or more methods. Typically, each pair of such methods should be executed. However, in the case polymorphism and dynamic binding the definite interclass relations can not be identified. They discuss the use of orthogonal array based technique to utilize possible interactions and create corresponding test cases. However, as the order of interactions increases the management of the technique becomes difficult.

### 3.2.7.4 Object Integration Testing

[Graham et al. 1993] examines the suitability of the V testing life cycle to object-oriented paradigm. The V testing life cycle integrates VV&T activities into the waterfall development lifecycle. The V testing lifecycle allows the development of the tests and model quality practices during the requirements specification, architectural design, module design and detailed design of the model development life cycle. During each phase listed, a corresponding test activity is scheduled; this approach ensures that testing is not left to the end of the implementation. After explaining the testing life cycle, the authors show that this approach is not completely suitable for object-oriented paradigm. Their analysis illustrates different levels of testing: acceptance and system testing, object integration testing, and class testing. While acceptance and system testing is considered to be similar to the conventional model testing, class testing and object integration testing is required to be different.

Class testing is performed by testing methods, testing inheritance, and analysis of method ordering strategies. They suggest using black-box and white-box testing for individual methods. They recommend an incremental strategy similar to the approach discussed in [Harrold at al. 1992] to reduce the number of test cases that need to be re-run during the testing of inheritance structure. For object integration testing an object instantiation tree that shows which objects can be instantiated from which others allow the tester to provide a top-down testing order. This strategy provides a plan for the integration of program units. Figure 6 shows a simple object instantiation tree for a composite class C1 which is the root of the tree. The integration testing process proceeds in a bottom-up manner: Classes C7 and C8 are instantiated and tested first. Then the branch of the tree including C6, C7, and C8 is tested. The integration testing continues incrementally until all the classes at the level below the root class is tested and integrated. Finally, the composite class with all its aggregate objects is tested.

The advantage of this approach is that independent branches can be tested in parallel at the same time. However, the instantiation tree assumes that the aggregation structure is a tree structure, whereas it can be a directed graph with cycles.

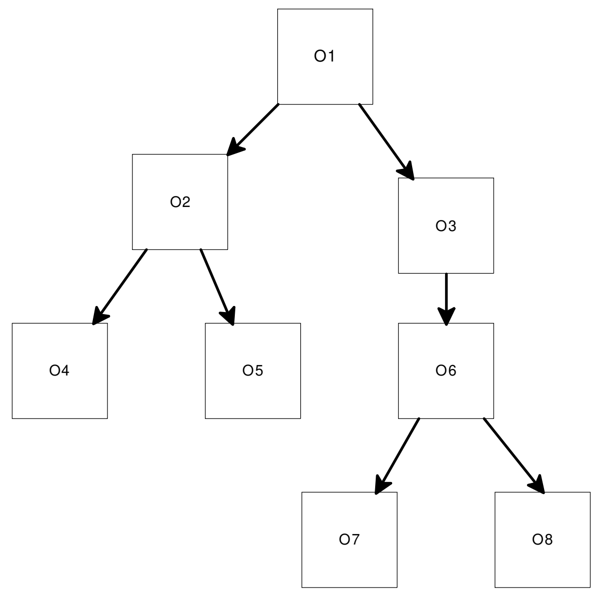


**Figure 6** Object Instantiation Tree

The authors propose an inter-object method ordering strategy during the integration testing using the object instantiation tree.

### 3.2.7.5 Propagation-Pattern Based Integration Testing

The set of objects needed for integration can be extracted from abstract model of class dependencies. Liberherr and Xiano [1992] discuss several specifications that can be used for integration test plan for object-oriented models. The specifications are abstract representations to facilitate reuse and evolution. Class Dictionary Graphs (CDG) provides the aggregation, association, and inheritance relationships among classes. Propagation patterns defines the responsibilities of the classes defined by Cads. They argue that implementations can be derived by traversing CDGs by propagation patterns. As the systems functionality is refined and evolved the CDGs are incrementally changed. They change history of CDGs is called evolution history. The authors suggest incremental testing based on CDGs and propagation patterns. Each component in the evolution history is tested incrementally with simple inputs to exercise parts of the propagation pattern it is associated with. Then the testing strategy becomes more complex by achieving total propagation pattern coverage. That is, at each iteration test sets that cover set of objects that exercise some or all responsibilities of propagation patterns are developed. They call the collective step-wise incremental coverage plan as growth-plan. The authors propose a strategy based on CDGs and propagation patters; however, they do not discuss the determination of test units, state of the objects in the test unit, the oracle problem which deals with the determination of expected results, and means for checking conformance to expected results.

#### 3.2.7.6 Wave Front Integration Testing

This technique is also advocated by McGregor and Korson [1994] in their testing process model. The authors suggest that the development of components in the system proceeds in parallel at the same time. At each iteration the developers reach an agreement about the functionality and the negotiated level of behavior for each component. Although this approach avoids the necessity for stubs, it requires more communication. The authors argue about extending contract-based design which defines the interfaces and collaborations among the classes. They propose adding a new section to the contract that specifies the deadlines to facilitate integration and testing schedules. This approach requires good communication and planning among the developers.

### *3.2.8 Method Sequence Testing*

While [Smith and Robson 1992] mentions the arbitrary application of methods to an object, on the contrary, [Kirani and Tsai 1994] provides a method sequence specification which defines the causal order of methods that an object should expect. Then this specification is used to test whether the implementation conforms to the specification. Their justification is based on the fact that the objects have state dependent behavior; that is, the result of any method execution depends on previously executed methods. Thus correct object behavior is possible if the methods are called in a well-defined manner. They argue that many of the inconsistent method sequences can be caught by statically analyzing control flow graph of the program. However, they agree that due to dynamic binding and polymorphism, static analysis of the program may not be able to identify all the sequences. In this case they suggest a run time verification system which monitors the execution of the program. The authors also show how method sequence specification can be used for test case generation and aid in developing a test coverage criteria. Then they show how to derive new sequence specification for derived classes automatically from the parent class' method sequence specification. Their approach not only provides means for verification and validation, but also outlines guidelines for ensuring the correctness of inheritance structure and the correctness of the interface of the object.

### *3.2.9 Modifier Sequence Testing*

Objects change their states as a result of the application of sequences of modifier methods. A modifier method is a method of a class which can alter the state of the class by modifying the attributes. Parrish et al. [1994] propose a technique that enumerates the sequence of method invocations at arbitrary depth using a tree structure. Test cases are developed by traversing the tree and sending the messages collected from the tree.

The authors also discuss the possibility of putting the object into desired state and then applying the next operation. However, their experiments show that breadth-first state generation approach catches errors more quickly than generating the states in the traces and then applying the next operation.

### *3.2.10 Modular Type/Subtype Verification*

Another paper describing the use of formal specifications for the verification of object-oriented programs is given by Gary T. Leavens [Leavens 1991]. In this approach the relationships among the ADTs are used as a method of reasoning about object-oriented software. Given two types A and B; if B is the subtype of A, then A is the supertype of B. Type B must provide similar behavior to that of A. Hence the supertype can be used in the reasoning about any of its subtypes. This reduces the level of verification that has to be performed. The author than emphasizes the importance of separating subtyping and subclassing.

Subtyping is defined to be a specification issue whose constraints are based on behavior; subclassing, on the other hand, is restricted by the inheritance of its parent's implementation details. The relationship between a type and subtype is subjected to a set of constraints to facilitate verification. That is, a subtype must satisfy the constraints of all its supertypes. The paper postulates that syntactic constraints applied to the specification ensures modular specification; however, it does not ensure modular verification. The author then describes the major problem with the modular verification of types and their subtypes. It is likely that a subtype's preconditions for a particular message will be different than the supertype's preconditions. He solves this problem by introducing a simulation relationship between the type of the actual arguments and the type of the arguments used during verification. The simulation relationship must preserve the validity of the assertions (preconditions and postconditions). The relationship must be constructed such that every object of a subtype simulates an object of its supertypes.

### *3.2.11 PACT Strategy for Component Testing*

McGregor and Kare [1996] develop a strategy and a corresponding framework for object-oriented component testing, called Parallel Architecture for Component Testing (PACT). They describe the framework by means of two pattern languages. Those patterns can be viewed as abstract descriptions of the framework and they facilitate the reuse of the proposed architecture. The architecture defines an organization and a process for component testing.

The first pattern language is used to create an infrastructure for component testing. It is composed of patterns which align component test strategy with the overall project, design a test architecture parallel to the software architecture, and create a set of abstract test classes.

The second pattern language is used to develop a component testing process and explains the PACT in detail. The architecture improves the observability and enhances the testability of the software. Moreover, describing the framework in terms of patterns in a language independent way, the author makes the architecture more usable and easy to apply.

Another pattern language for object-oriented systems testing is proposed by Firesmith [1996].

### *3.2.12 State Testing*

#### 3.2.12.1 Graph-Based Class Testing

Hoffman and Strooper [1993] develop an automated state testing technique using a state model of the class under test. The state model is represented by a directed graph, called testgraph, and is represented by a driver class. The state of the class under test is compared to the expected state in the testgraph using an oracle class. The oracle class is similar to the class under test, but with less number of states and transitions. That is, it has the same operations of the class under test, but includes only those states and transitions included in the testgraph. The test execution proceeds on both the class under test and the testgraph class.

The test sequences are obtained from the testgraph class and then the results of the execution are checked using the oracle class. The testgraph is a directed graph in which the nodes represent the states and the arcs represent the transitions. The test sequences are traces of method calls obtained from the test graph. Although the state space of the class under test is large, the state space of the testgraph is smaller since it contains only the states that can be reached by the test suite.

#### 3.2.12.2 Object State Testing

The object state testing technique, discussed in [Kung et al. 1993; Kung et al. 1994; Kung et al. 1995b; Gao et al. 1995], is also a state testing technique. The OSD diagram is used for object state testing. For each instance variable an Atomic Object State Diagram (AOSD) is constructed. The states are identified in terms of the disjoint intervals of instance variable's domain. Then a Composite Object State Diagram (COSD) is constructed by combining the AOSDs. The process of generating test cases from the COSD is similar to Chow's [1978] technique. A test tree whose nodes are composite states of COSD is constructed. The edges of the tree represent the transitions between the composite states. If the COSD contains n AOSDs, then each state is identified by an n-tuple where the i-th component of the tuple represents the state of the i-th AOSD. The details of the algorithm for the construction of a COSD, and generating test cases from the composite diagram are described in [Kung et al. 1994] of bindings to test.

#### 3.2.12.3 State-Based Functional Testing

A state-based approach is given in [McGregor and Dyer 1993]. The test strategy relies on the development of a state model of a class from OOA/D specification or by examination of the implementation of the class under test. The authors use a graphical representation similar to statecharts. The approach also requires the use of strict inheritance to facilitate the reuse of state-based test cases of superclasses. The test strategy proceeds in several stages: (1) testing of each constructor method, (2) testing of each accessor method, (3) developing test cases for checking the post-condition for each method, (4) test each transition in the state model of the class (i.e., transition testing). Each class keeps an invariant and each test case checks whether the current state complies to the invariant. They argue that due to polymorphism and dynamic binding, possible number of client objects and polymorphic servers can be large and they propose the use of orthogonal arrays to facilitate selection.

#### 3.2.12.4 State-Based Testing

Turner and Robson [1993b] propose a new technique, called state-based testing, for the validation of object-oriented systems. The authors argue that majority of object-oriented testing research do not address the differences between procedural and object-oriented paradigm. They mention that due to the interactions between the features and the state of the object, new techniques are necessary. Their approach defines the state of an object as the combined values from all of its data members. The substate is the value of a single data member at the current point in time. Then a partial state is defined as the combination of subset of all the data members in the class. Since the range of values that a data member is large, the authors propose two types of substate-values: specific substate values - are those values which have special significance, and general substate values - are those values which can be considered in the same manner by the program.

The state space becomes manageable by dividing it into specific and general sections. This notion of restricting the state space is initially explored in [Bernot et al. 1991] as a hypothesis (regularity hypothesis) to restrict the exhaustive test set into a manageable size. Once the states are identified, the methods and the input and output states are examined. That is, the input states for which the method is applicable and the output states which result by the application of the method are identified. Then test cases are generated to create the initial condition (starting state for a method) along with the code which validates the starting state. After this, the method is applied and the resulting state is checked to see the conformance to the expected final state.

Turner and Robson's technique [1993b], state-based testing, uses Finite State Machine (FSM) as a test model and has the difficulty of representing the inherited state-dependent behavior, concurrent dynamic object behavior. Object state testing is based on a test model, called OSD (Object State Diagram), which extends the notion of FSM and incorporates both encapsulated and inherited state hierarchy. However, since possible state interactions must be reconstructed from the analysis of groups of state diagrams, object state testing is error prone and complex. On the other hand, because of the simplicity of FSM representation and its well known principles, state-based testing is both easy to learn and use.

## 4. COST-BENEFIT INDICATORS

We identified several indicators to characterize the effectiveness, benefit, and costs of the reviewed techniques. The indicators are as follows:

4. *Error detection capability* emphasizes the faults that can be uncovered by the technique.
5. *Level of formality* indicates the relative level of mathematical formality and reasoning capability of the technique.
6. *Complexity* indicator concerns with the complexity of the technique. This indicator is characterized by the following two subindicators.
    - *Ease-of-learning* indicates the ease of the understandability and applicability of the technique by the model testers.
    - *Ease-of-running/interpretation* indicates the ease of interpreting the outcome of the test.
7. *Usage* indicates the extent of usage and the commonality of the application of the technique.
8. *Set-up/computer resource cost* indicates the cost with respect to computer resources and testing environment.

## 5. ASSESSMENT OF THE EFFECTIVENESS OF THE VV&T TECHNIQUES

To assess the effectiveness of the techniques we present a preliminary fault taxonomy which is compiled through a comprehensive literature survey on object-oriented testing literature and the authors' experience. The faults uncovered by the techniques indicate their error detection power. The error detection power along with the other indicators provide a basis for comparing the techniques.

Section 5.1 presents the model fault taxonomy and Section 5.2 the subjective values of indicators assigned to the techniques.

### 5.1 A Preliminary Object-Oriented Model Fault Taxonomy

We compiled a preliminary set of faults as a result of comprehensive literature survey and our experience in the V&V field. Six categories of faults are identified: (1) class/object faults, (2) message faults, (3) attribute faults, (4) method faults, (5) inheritance faults, (6) integration faults. Faults identified in [Firesmith 1993; Purchase and Winder 1991] along with the additional faults applicable to modeling are combined to create this fault taxonomy. Figure 7 illustrates the fault classification and related faults and anomolies.

### 5.2 Comparative Analysis of VV&T Techniques

In this section we analyze each V&V technique included in the taxonomy with respect to the effectiveness and cost indicators discussed in Section 4. The analysis is based on the subjective interpretation of the author based on his experience and knowledge related to the techniques.

**Figure 7** Preliminary Fault/Anomoly Taxonomy

**Table 2.** Comparative Analysis of Object-Oriented VV&T Techniques

| VV&T Technique | Error Detection Capability | Formality | Learning and Human Resource | Interpretation | Usage | Resource Cost |
|---|---|---|---|---|---|---|
| Adequacy Crriteria-Based Testing | 2.1-3, 6.1-7 | Informal to Formal | Easy | Easy | High | High |
| Class Testing by Pointer Examination | 4.7, 6.9 | Formal | Difficult | Difficult | Low | High |
| Data Flow Testing | 1.9, 3.1, 6.2 | Formal | Moderate | Moderate | High | Moderate |
| Domain Dependent Testing | 1.1-2, 1.6, 1.8, 1.10, 5.4-9 | Informal | Moderate | Easy | High | Low |
| Fault-Based Testing | Identified fault list | Informal | Easy | Easy | High | Low |
| Hypothesis-Based Testing | 1.3-4, 2.2, 3.3, 4.3 | Formal | Moderate | Easy | Moderate | Moderate |
| Modal-Based Testing | 1.1, 2.2, 3.1, 4.6 | Formal | Moderate | Easy | High | Low |
| Three-View Model Testing | 1.3, 1.5, 1.7, 1.8, 1.10, 2.4, 2.6, 3.3, 5.4-5.9 | Informal | Moderate | Easy | High | Low |
| Algebraic Specification-Based Testing | 1.3, 1.5, 4.3, 4.7 | Very Formal | Moderate | Moderate | Moderate | High |
| Class Firewall Technique | Inapplicable | Formal | Moderate | Easy | Low | Moderate |
| Component Certification Technique | 1.3, 2.2, 6.8 | Very Formal | Moderate | Moderate | Moderate | High |
| Free Approach | 1.3, 1.5, 2.1, 2.2, 6.5-7 | Formal | Moderate | Moderate | High | Moderate |
| FOOT Testing Approach | 2.2, 4.7, 5.3 | Less Formal | Easy | Easy | Moderate | Low |
| Inheritance Testing | 5.1-10 | Formal | Moderate | Easy | Moderate | Moderate |
| Integration Testing | 2.1, 2.3-6, 6.1-7 | Informal to Formal | Moderate | Easy to Moderate | High | Low to High |
| Method Sequence Testing | 1.3, 1.5, 2.2 | Formal | Easy | Easy | Moderate | Moderate |
| Modifier Sequence Testing | 1.3, 1.5, 2.2 | Formal | Easy | Easy | Moderate | Moderate |
| Modular Type/Subtype Verification | 5.3 | Very Formal | Difficult | Difficult | Low | Low |
| PACT Strategy for Component Testing | 1.3, 1.4, 3.3, 4.1-7 | Formal | Easy | Easy | High | High |
| State Testing | 1.1, 1.3-5, 2.2, 3.3, 4.7 | Formal | Moderate | Easy | High | High |

## 6. CONCLUSIONS

Although object-oriented paradigm (OOP) is known for decades, it is not used extensively for large scale simulation model development until recently. The transition to OOP is not easy due to the difficulty of managing the iterative, incremental and evolutionary development. As complex simulation models are increasingly developed using OOP, a new problem emerges: how do we verify and validate a complex object-oriented simulation model and what is the best strategy that can cope with the problems discussed in this paper?

A taxonomy of V&V techniques for object-oriented simulation models is presented. These techniques have recently been proposed in the object-oriented software testing literature. The ones that are applicable for object-oriented simulation models are included in the taxonomy.

Most of the existing techniques focus on unit (class) level V&V and do not take the interactions and diverse patterns of interactions among objects into account. That is, there is a lack of well defined integration level V&V techniques in the literature. Those techniques that mention integration level V&V consider a structural point of view (i.e., the interrelationships such as aggregation and instantiation order) and propose top-down or bottom-up testing strategies.

Simulation and modeling is moving towards greater use of components to develop large scale simulation models. The development of component-based models introduces fundamental changes in the way models are acquired, integrated, deployed, and evolved. Rather than the classic development life cycle, models are designed by qualifying existing components followed by an iterative process of integrating and assembling the components. The recent advancements in object-oriented development provides a basis in which model components are integrated and assembled as basic building blocks of the model. Based on the initial investigations on object-oriented model V&V, we are currently researching the V&V issues regarding the component-based object-oriented simulation models. The traditional V&V approaches have been mainly concerned with the functionality of components. Particularly, our V&V research aims to elevate component relationships to the same level of concern. The interaction, coordination, organization, and cooperation of the components are the major issues to be investigated.

## REFERENCES


Adrion Richards W., M. A. Branstad, J. C. Cherniavsky (1982). "Validation, Verification, and Testing of Computer Software," Computing Surveys 14, 2 (June), 159-192.

Balci, O. (1997). “Verification, Validation and Testing”, In *The Handbook of Simulation*. J. Banks, Editor, John Wiley & Sons, New York, NY, to appear.

Balci, O.; P. Glasow; P. Muessig; E. Page; J. Sikora; S. Solick; and S. Youngblood. (1996). *Department of Defense Verification, Validation and Accreditation (VV&A) Recommended Practices Guide*, Defense Modeling and Simulation Office, Alexandria, VA, Nov.

Barbey, S. and A. Strohmeier. (1994). “The Problematics of Testing Object-Oriented Software.” In *Proceedings of Software Quality Management: Vol. 2, Building Quality into Software* (Edinburg, UK, July 26-28). Computational Mechanics Publications, Southampton, UK, 411-425.

Barbey S. and A. Strohmeier (1994). "The Problematics of Testing Object-Oriented Software," Software Quality-Management II Vol 2: Building Quality into Software, Computational Mechanics Publications, Boston, 411-425.

Barbey S. (1995). "Testing Ada 95 Object-Oriented Programs," Proceedings of the 2nd International Eurospace-Ada-Europe Symposium, LNCS 1031, Springer Verlag New York. 406-418.

Beizer B. (1990). Software Testing Techniques, 2nd ed, Van Nostrand-Reinhold Company, New York, NY.

Berard E. (1992). Essays on Object-Oriented Software Engineering, Volume 1, Prentice-Hall, Englewood Cliffs, NJ.

Bernot G., M. C. Gaudel, and B. Marre (1991). "Software Testing Based on Formal Specifications: A Theory and A Tool," Software Engineering Journal, November, 387-405.

Binder R. (1994a). "Testing Object-Oriented Systems: A Status Report," American Programmer 7,4 (April), 22-28.

Binder R. (1994b). "Design for Testability with Object-Oriented Systems," Communications of the ACM 37, 9 (September), 87-101.

Binder R. (1995). "Trends in Testing Object-Oriented Software," IEEE Computer, October, 68-69.

Binder R. (1996). "Modal Testing Strategies for OO Software," IEEE Computer, November, 97-99.

Booch G. (1991). Object-Oriented Design with Applications, Benjamin Cummings, Menlo Park, CA.

Booch G. (1994). Object-Oriented Analysis and Design with Applications, Benjamin Cummings, Redwood City.

Cheatham J. T. and L. Mellinger (1990). "Testing Object-Oriented Software Systems," Proceedings of the 1990 ACM 18th Annual Computer Science Conference, ACM Press, New York. 161-165.

Chen Y. T. and C. K. Low (1995). "Dynamic Data Flow Analysis for C++," Proceedings of the Asia-Pacific Software Engineering Conference, December, IEEE Computer Society Press, Los Alamitos, CA. 22-28.

Chow S. T. (1978). "Testing Software Design Modeled by Finite-State Machines," IEEE Transactions on Software Engineering 4, 3 (May), 178-187.

Chung C.-M. and M.-C. Lee (1992). "Object-Oriented Programming Testing Methodology," Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering, June, IEEE Computer Society Press, Los Alamitos, CA. 378-385

Coad P. and E. Yourdon (1991). Object-Oriented Analysis, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ.

D'Souza J. R. and R. J. LeBlanc (1994). "Class Testing by Examining Pointers," Journal of Object-Oriented Programming 7, 4 (July), 33-39.

Doong R.-K. and P. Frankl (1994). "The ASTOOT Approach to Testing Object-Oriented Programs," ACM Transactions on Software Engineering and Methodology 3, 4 (April), 101-130.

Firesmith G. D. (1993). "Testing Object-Oriented Software," Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems(TOOLS USA'93), Prentice-Hall, Englewood Cliffs, NJ. 407-426.

Firesmith G. D. (1996). "Pattern Language for Testing Object-Oriented Software," Object Magazine 5, 9 (January), 32-38.

Fishwick A. P. (1995). *Simulation Model Design and Execution*. Prentice-Hall, Englewood Cliffs, NJ.

Gannon J., P. McMullin, R. Hamlet (1981). "Data Abstraction Implementation, Specification, and Testing," ACM Transactions on Programming Languages and Systems 3, 3 (July), 211-223.

Gao Z. J., D. Kung, P. Hsia, Y. Toyoshima, and C. Chen (1995). "Object State Testing for Object-Oriented Programs," Proceedings of the 19th International Computer Software & Applications Conference, IEEE Computer Society, Los Alamitos, CA. 232-238.

Graham A. J., A. C. T. Drakeford, and C. D. Turner (1993). "The Verification, Validation, and Testing of Object-Oriented Systems," BT Technology Journal 11, 3. 79-88.

Harrold J. M., J. D..McGregor, K. J. Fitzpatrick (1992). "Incremental Testing of Object-Oriented Class Structures," Proceedings, 14th International Conference on Software Engineering, May, IEEE Computer Society, Los Alamitos,CA. 68-80.

Harrold J. M. and G. Rothermel (1994). "Performing Data Flow Testing on Classes," Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, December, ACM Press, New York. 154-163.

Hayes H. J.(1994), "Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach," Proceedings, Object-Oriented Methodologies and Systems, E. Bertino, S. Urban (Eds.), LNCS 858. Springer Verlag, Berlin. 205-220.

Hill, R.C.D. 1996. *Object-Oriented Analysis and Simulation*. Addison-Wesley, Reading, MA.

Hoffman D. (1989). "A Case Study in Module Testing," Proceedings of the Conference on Software Maintenance, October, IEEE Computer Society Press, Washington D.C., 100-105.

Hoffman D. and Strooper P. (1993). "Graph-Based Module Testing," Proceedings of the Sixteenth Australian Computer Science Conference, (February), 479-487.

Hughes M. and D. Stotts (1996). "Daistish: Systematic Algebraic Testing for OO Programs," Software Engineering Notes, 21, 3 (May), 53-61.

Jacobson I. (1992). Object-Oriented Software Engineering: A Use-Case Driven Approach, ACM Press/Addison Wesley, New York, NY.

Jorgensen C. P. and C. Erickson (1994). "Object-Oriented Integration Testing," Communications of the ACM 37, 9 (September), 30-38.

Kandelin A. N. and D. E. O'Leary (1995). "Verification of Object-Oriented Systems: Domain Dependent and Domain-Independent Approaches," Journal of Systems and Software 29, 261-269.

Kirani S. and W. T. Tsai (1994). "Method Sequence Specification and Verification of Classes," Journal of Object-Oriented Programming 7, 6 (October), 28-38.

Korson T. and J. D. McGregor (1990). "Understanding Object-Oriented: A Unifying Paradigm," Communications of the ACM 33,9 (September), 40-60.

Kung David J., J. Gao, P. Hsia, Y. Toyoshima, and C. Chen (1993). "Design Recovery for Software Testing of Object-Oriented Programs," Proceedings of the Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos, CA. 202-211.

Kung David J., N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen (1994). "On Object State Testing," Proceedings, The 18th International Computer Software & Applications Conference, IEEE Computer Society Press, Los Alamitos, CA. 222-227.

Kung David J., J. Gao, P. Hsia, J. Lin, and Y. Toyoshima (1995a). "Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs," Journal of Object-Oriented Programming 8, 2 (May), 51-65.

Kung David J., P. Hsia, Y. Toyoshima, C. Chen, Y. Kim and Y. Song (1995b). "Developing an Object-Oriented Software Testing and Maintenance Environment," Communications of the ACM 38, 10 (October), 75-86.

Lalioti V. and P. Loucopoulos (1993). "Visualization for Validation," Advanced Information Systems Engineering. Proceedings of the 5th International Conference, CAISE'93, June, Springer Verlag. 143-164.

Leavens T. G. (1991). "Modular Specification and Verification of Object-Oriented Programs," IEEE Software, 8, 4 (July), 72-80.

Lee J. W. (1987). "Software Testing and Verification", In Advances in Computers, 26, 335-391.

Lieberherr K. J. and C. Xiao (1992). "Object-oriented software evolution," IEEE Transactions on Software Engineering, 19, 4, 313-343.

Linos K. P. and V. Courtois (1994). "A Tool for Understanding Object-Oriented Program Dependencies," Proceedings of the Workshop on Program Comprehension, November, 20-27.

Martin J. and J. J. Odell (1992). Object-Oriented Analysis & Design, Prentice-Hall, Englewood Cliffs, NJ.

McCabe J. T. and A. H. Watson (1994). "Combining Comprehension and Testing in Object-Oriented Development," Object Magazine 4, 1 (March-April), 63-66.

McGregor D. J. and Dyer D. M. "A note on inheritance and state machines," Software Engineering Notes, 18, 4 (October), 61-69.

McGregor D. J. and T. Korson (1994). "Integrating Object-Oriented Testing and Development Processes," Communications of the ACM, 37, 9 (September), 59-77.

McGregor D. J. and A. Kare (1996). "Parallel Architecture for Component Testing of Object-Oriented Software," Proceedings, 9th Annual Software Quality Week, May, Software Research, Inc. San Francisco.

Overbeck J. (1994). "Integration testing for object-oriented software," Ph.D. dissertation, Vienna University of Technology, Vienna, Austria.

Pande H., W. Landi, B. G. Ryder (1994). "Interprocedural def-use associations in C programs," IEEE Transactions on Software Engineering, 20, 5 (May), 385-403.

Parrish A., R. B. Borie, D. W. Cordes (1993). "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," Journal of Systems and Software 23, 2 (November), 95-109.

Parish A. S., D. Cordes, M. Govindarajan (1994). "Systematic defect removal from object-oriented modules," Proceedings of the 7th International Software Quality Week, May, Software Research, Inc. San Francisco.

Perry E. D. and G. E. Kaiser (1990). "Adequate Testing and Object-Oriented Programming," Journal of Object-Oriented Programming, 2, 5 (January/February), 13-19.

Poston M. R. (1994). "Automated Testing from Object Models," Communications of the ACM 37,9 (September), 48-58.

Purchase A. J. and R. L. Winder (1991). "Debugging Tools for Object-Oriented Programming," Journal of Object-Oriented Programming 4, 3 (June), 10-27.

Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ.

Shlaer S. and S. Mellor (1988). Object Lifecycles: Modeling the World in States, Prentice-Hall, Englewood Cliffs, NJ.

Smith D. M. and D. J. Robson (1990). "Object-Oriented Programming: The Problems of Validation," Proceedings of the 6th International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA. 272-282.

Smith D. M. and D. J. Robson (1992). "A Framework for Testing Object-Oriented Programs," Journal of Object-Oriented Programming 5, 3 (June), 45-53.

Subramanian S., W. T. Tsai, S. H. Kirani (1994). "Hierarchical Data Flow Analysis for O-O programs," Journal of Object-Oriented Programming, May, 36-46.

Turner D. C. and D. J. Robson (1993a). The Testing of Object-Oriented Programs. Technical Report TR-13/92. Durham, England: Computer Science Division, School of Engineering and Computer Science (SECS), University of Durham (UK).

Turner D. C. and D. J. Robson (1993b). "The State-Based Testing of Object-Oriented Programs," Conference on Software Maintenance, September, IEEE Computer Society Press, Los Alamitos, CA. 302-310.

Voas M. Jeffrey (1996). "Object-Oriented Software Testability," Achieving Qaulity in Software. Proceedings of the 3rd International Conference on Achieving Quality in Software, Chapman & Hall, London. 279-290.

Wirfs-Brock R., B. Wilkerson, and L. Wiener (1990). Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, NJ.

Wohlin C. and P. Runeson (1994). "Certification of Software Components," IEEE Transactions on Software Engineering 20, 6 (June), 494-499.

## AUTHOR BIOGRAPHY

**LEVENT YILMAZ** is a Ph.D student in the Department of Computer Science at Virginia Tech. He received a B.S degree in Computer Engineering and Information Science from Bilkent University in 1993 and an M.S. degree in Computer Science from Virginia Tech in 1996. His current Ph.D. research focuses on the verification, validation and testing of object-oriented simulation models. Mr. Yilmaz is a member of ACM and IEEE Computer Society.