On Object State Testing*

D.C. Kung, N. Suchak, J. Gao, P. Hsia
The University of Texas at Arlington
P.O. Box 19015, Arlington, TX 76019-0015
Y. Toyoshima, C. Chen
Fujitsu Network Transmission Systems, Inc.

February 26, 1994

Abstract

The importance of object state testing is illustrated through a simple example. We show that certain errors in the implementation of object state behavior cannot be readily detected by conventional structural testing, functional testing, and state testing. We describe an object state test model and outline a reverse engineering method for extracting object state behaviors from C++ source code. The object state test model is a hierarchical, concurrent, communicating state machines. It resembles the concepts of inheritance and aggregation in the object-oriented paradigm, rather than the concept of state decomposition as in some existing models. The reverse engineering method is based on symbolic execution to extract the states and effects of the member functions. The symbolic execution results are used to construct the state machines. The usefulness of the model and the method is discussed in the context of object state testing in the detection of a state behavior error.

1 Introduction

Object state testing is an important aspect of object oriented (OO) software testing. It is different from control flow and data flow testings. In control flow testing, the focus is testing the program according to the control structures, i.e., sequencing, branching, and iteration. In data flow testing, the focus is testing the individual data define-and-use relationships. Object state testing focuses on objects' state dependent behavior rather than the control structures and individual data.

Object state testing should address a number of challenging problems, which have not been systematically tackled in the literature. That is, we need to develop 1) a suitable state test model; 2) a methodology for constructing a state test model, from either a specification or a given OO program; 3) state testing strategies and testing criteria; and 4) methods and techniques for test case generation and result evaluation.

In the last two years, we have worked on the development of a comprehensive framework for OO software testing [18] [19] [20] [21]. This includes a formal test model for OO programs, a methodology, an OO test environment, and supporting databases. The work described in this paper is part of this effort and deals with automatic construction of object state behaviors from OO programs for program-based object state testing. In particular, we describe an object state test

[0]In Proceedings of IEEE COMPSAC'94 Conference.

model, called *object state diagram (OSD)* to capture state dependent behaviors of the objects. We then informally present a methodology for extracting object state behavior from an OO program written in C++. The usefulness of the OSD in testing will also be discussed.

We wish to point out that the OSD is defined with testing rather than modeling in mind; that is, it is defined to facilitate a tester to understand the state dependent behaviors of the object classes, to generate state test cases, and to evaluate the test results.

The state behavior extraction methodology is based on symbolic execution [6] to identify the states of the data members and the effects of the member functions on the states of the data members. It is well-known that the complexity of symbolic execution remains to be solved except for applications to simple functions. Fortunately, most member functions in an OO program consist of only a few statements [30]. Our experience shows that it works very well for such functions.

The importance of the work can be stated as follows. First, object state testing has not received wide attention and in-depth treatment, although state machines have been used to test conventional [5] and protocol software [17]. Second, symbolic execution has traditionally been used to prepare test cases for testing a single function, not interactions between the member functions of a class through object state. Third, although integration testing of an OO application program may detect some state behavior errors, the difficulty and cost will be much higher than detecting (and removing) such errors at class level testing. Moreover, the advantage of class library and software reuse will be largely compromised if the classes are not well tested.

The organization of this paper is as follows. Section 2 provides a brief review of basic problems and existing solutions in software state testing. Also discussed are new problems introduced by the OO features and some existing solutions. In section 3, we discuss the importance of object state testing through a simple example. In section 4, we introduce a state test model for capturing and representing object state dependent behavior. In section 5, we present the methodology for extracting the state dependent behavior from C++ code. In section 6, we discuss the usefulness of the OSD in testing and in section 7, we present the conclusions and future work.

2 Related Work

State dependent behaviors of a software system are usually modeled by finite state machines (FSM) [10]. In particular, Ward and Mellor used FSM to specify the state dependent behaviors of the control processes in real time structured analysis and design [29]. Booch [2], and Coad and Yourdon [7] used FSM to model the state behaviors of objects. Unlike Booch, and Coad and Yourdon, Rumbaugh et al [24] used nested, concurrent FSM instead of the conventional flat FSM. Improvements were later introduced by Coleman et al in [8]. It should be pointed out that nested, concurrent state machines, called Statemate, were first introduced and systematically studied by Harel [13] [14] to model reactive systems. Nesting reduces the complexity whereas concurrency significantly increases the modeling power.

FSM are extensively used in protocol verification, validation, and testing, see e.g., [17] [22] [25] [28]. Its use as a software test model is found in [1] and [5]. State testing is considered complementary to functional and structural testings [1]. A distinguished approach to state testing was due to Chow [5] in which FSM is used to model software components that can be described by *stimuli* and *operations*. Stimuli are inputs from the outside world and operations are executed as responses (to the stimuli). In Chow's approach, a spanning tree derived from an FSM specification of software

is used to generate test sequences, each of which is a sequence of operations. These test sequences can be used to detect errors in operations, state transitions, and extra and missing states.

Software testing of OO programs is a relatively new area. Discussions of some OO testing problems can be found in [4] [26] [30]. Perry and Kaiser [23] discussed the necessity of retesting certain classes/member functions in the context of reuse. Harrold et al [15] proposed a methodology that provides strategies to test an inheritance hierarchy and test case reuse. Our previous work on OO testing proposed a reverse engineering approach to extract class relations, member function interfaces, and control flow and data flow information from C++ code. This information is used to generate test strategies and prepare test cases [18]. Recently, we have extended the reverse engineering model to reduce retesting effort in the maintenance phase [20].

OO state testing is an important and difficult area of OO testing. There is almost no report in the literature that is specifically devoted to this topic. The state test model presented in this paper is superficially similar to existing models [14] [24] [3]. However, the semantics and use are completely different. The existing models are used as analysis and design tools, but OSD is used as a testing tool, i.e., a program-based state test tool. The notions of state and transition in OSD are associated with specific programming concepts rather than high level application domain concepts. For example, a state of an object is defined by the ranges of values of a subset of member data of the object. State transitions are defined by (possibly conditional) execution of a member function. More importantly, the hierarchy of state machines in our approach resembles the inheritance and aggregation hierarchies of object classes, whereas the existing concept of state hierarchy resembles the decomposition of complex states into a network of simpler states.

OO state testing as presented in this paper concerns multiple, hierarchical, concurrent, communicating state machines, constructed from code. It is different from Chow's method [5], where test cases are generated from a single state machine; and hence, interactions between the objects are not considered. We shall show in section 6, testing of such interactions is essential in the detection of state behavior errors.

3 Object State Testing

Conventional functional and structural testings may detect errors that are local to a member function. They are not adequate for detecting errors due to interactions between member functions through object state behaviors. In this section, we show informally the notion of object state and the importance of object state testing through a simple example.

3.1 A Motivating Example

Consider a coin box of a vending machine implemented in C++. For simplicity, we will assume that the coin box has very simple functionality and the code to control the physical device is omitted. It accepts only quarters and allows vending when two quarters are received. It keeps track of the total quarters (denoted totalQtrs) received, the current quarters (denoted curQtrs) received, and whether vending is enabled or not (denoted allowVend). Its functions include adding a quarter, returning the current quarters, resetting the coin box to its initial state, and vending. The C++ source code for this simple coin box is listed below:

```
class CCoinBox
```

```
{
  unsigned totalQtrs; // total quarters collected
  unsigned curQtrs; // current quarters collected
  unsigned allowVend; // 1 = vending is allowed
 public:
  CCoinBox() { Reset(); }
  void AddQtr(); // add a quarter
  void ReturnQtrs() { curQtrs = 0; } // return current quarters
  unsigned isAllowedVend() { return allowVend; }
  void Reset() { totalQtrs = 0; allowVend = 0; curQtrs = 0; }
  void Vend(); // if vending allowed, update totalQtrs and curQtrs
};

void CCoinBox::AddQtr()
{
  curQtrs = curQtrs + 1; // add a quarter
  if (curQtrs > 1) // if more than one quarter is collected,
    allowVend = 1; // then set allowVend
}

void CCoinBox::Vend()
{
  if ( isAllowedVend() ) // if allowVend
  {
     totalQtrs = totalQtrs + curQtrs; // update totalQtrs,
     curQtrs = 0;                     // curQtrs, and
     allowVend = 0;                   // allowVend,
  }                                   // else no action
}
```

By carefully examining the code above one may discover that there is an error in the implementation. But the error is not obvious. We argue that the error cannot be easily detected by functional testing and/or structural testing of the member functions since: 1) each of the member functions seems to implement the intended functionality; 2) structural testing (such as basis path testing) also would conclude that each basis path is correctly executed; and 3) the most important is that the error was due to interactions involving more than one member function through an object state. Such interaction implies that testing the individual functions is not likely to discover the error.

3.2 Error Detection via State Testing

In our opinion, the states of an object are defined by a subset of the member data of the object class. For the coin box example, the subset of member data that define the states of the coin box are allowVend and curQtrs, because these two member data are checked in a conditional statement to determine whether certain actions need to be performed. For example, the allowVend is checked in Vend() to determine if totalQtrs, curQtrs, and allowVend should be updated. Similarly, curQtrs is checked in AddQtr() to set allowVend. TotalQtrs is not a state defining member datum because the object behaves independently from changes to this datum.

From the two possible outcomes of the test of allowVend we know that there are two possible intervals or states: $[0, 0], [1, M]$, where M denotes the maximum possible value of unsigned for a particular implementation. Similar, there are two states for curQtrs: $[0, 0], [1, M]$. Thus, there are totally four states for the coin box, as illustrated in Figure 1.

As usual, an object changes its state when a member function is executed. The effects of the

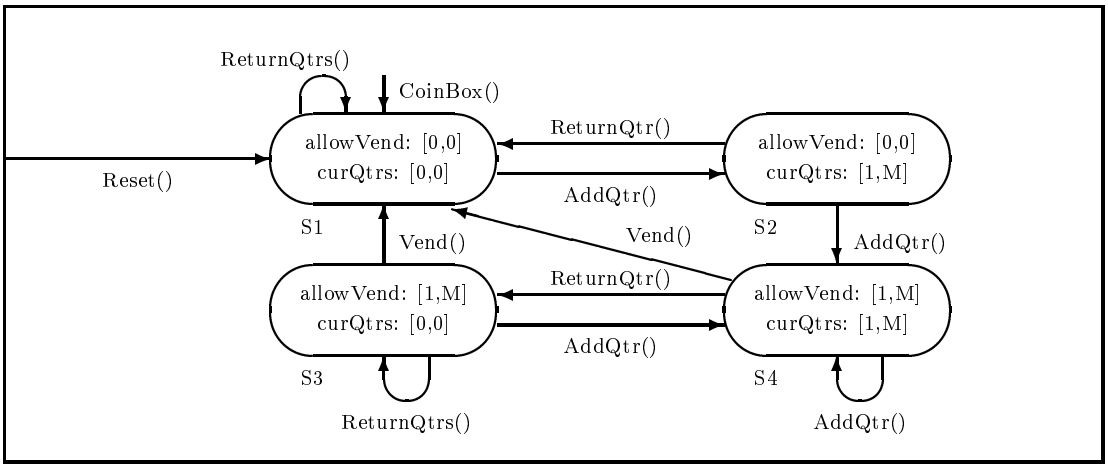


Figure 1: States and state transitions for the coin box example

member functions can be analyzed by identifying the pre-state in which a member function can be applied, and the post-state that results. For example, the AddQtr() function can be applied to any of the four states. Its application in state S1 results in state S2, its application in state S2 results in state S4 in which allowVend is set to greater than zero due to the successful testing of the condition "curQtrs + 1 > 1" (which is equivalent to "curQtrs > 0"). The analysis of all of the functions results in the state transition diagram in Figure 1. State S1 is an initial state since it is the resulting state of a constructor. Note that the pre-state of a constructor is not shown to conceptually indicate that the object is created from scratch. Note also that the transition labeled by Reset() connects the enclosing rectangle and state S1. This is a shorthand notation to mean that the Reset() member function can be executed in any of the states and the execution always results in state S1. The member function isAllowVend() is not shown for simplicity.

From the state transition diagram, we see that the following scenario or member function sequence can occur: AddQtr(); AddQtr(); Vend(). The sequence constitutes a normal and expected behavior of the coin box. However, the diagram also indicates that the scenario AddQtr(); AddQtr(); ReturnQtrs(); Vend() is also possible. This means that a consumer could insert two quarters, instruct the coin box to return the quarters, and Vend() to get a free drink. A state dependent error is detected.

The error is caused by not resetting the member datum allowVend to zero when the ReturnQtrs() member function is executed. It can be removed by changing ReturnQtrs() to

$$void \quad ReturnQtrs(\{curQtrs = 0; allowVend = 0; \}$$

It is important to point out that there are many possible ways to implement the correct semantics. This makes it difficult to detect such errors by conventional functional and structural testings. For example, there could be a disAllowVend() member function, which could be invoked to set allowVend to zero, and Vend() would test both allowVend and curQtrs to determine whether vending is allowed. Thus, ReturnQtrs() would not be changed. Although this latter alternative may not be elegant, but it could exist in practice.

3.3 The Complexity of the Flat State Machine

The state machine in Figure 1 is called a flat state machine since the states and transitions are represented in only one level. Flat state machines are commonly used in modeling and testing function-oriented software. However, our experience showed that flat state machines are not appropriate for modeling and testing in the OO paradigm because these machines tend to introduced excessive complexity. Suppose a class has k state defining member data, each has $n_i, i = 1, \ldots, k$ value intervals (e.g., curQtrs has two intervals: [0,0] and [1,M]). The total number of possible combined states will be $N = n_1 * n_2 * \ldots * n_k$. In the worst case, there could be $N * N = n_1^2 * n_2^2 * \ldots * n_k^2$ directed edges, each of which may be labeled by more than one member function since two member functions may cause the same transition. For the coin box example above, we have four states and in the worst case there can be as many as 16 directed edges representing the state transitions.

4 The Object State Test Model

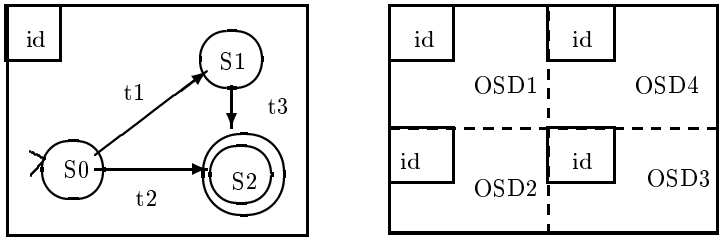
A state of a simple object in C++ is captured by ranges of values of a subset of member data of the object. For instance, a library book is available for check out if its number_of_copies is not equal to zero. In this example, the member data number_of_copies defines the state of book, while the member data book_title does not participate in the definition of the object's state because state changes of the object are independent of the member data.

The complexity of using the conventional flat state machines, as discussed in the last section, suggests that a hierarchical, concurrent object state diagram be used to represent the object state behavior. That is, the state behavior of a complex object is viewed as consisting of three parts: 1) the inherited part; 2) the aggregated part; and 3) the defined part. The inherited part consists of state machines of the base classes from which the complex object is derived. Similarly, the aggregated part consists of state machines of the component classes that are part of the complex object. The defined part consists of state machines for the state defining member data (e.g., number_of_copies) of the object. In particular, there is a state machine for each state defining member datum. Our experience showed that the introduction of hierarchy simplifies representation and analysis.

The general representation of the state behavior of a complex object is illustrated in Figure 2.

Stating more formally, an atomic OSD, denoted AOSD, for a class C is a deterministic finite state machine $AOSD = (S, \sigma, \delta, q_0, q_f)$ where

- S is a finite set of states, each of which is either a simple condition or an interval of data values of a (state defining) member datum defined in class C;
- σ a finite set of triggers, each of which consists of a (possibly empty) guard condition, a member function, followed by a (possibly empty) sequence of member functions (called responses);
- $q_0 \subseteq S$ is the set of initial states;
- $q_f \subseteq S$ is the set of final states; and
- δ a mapping from $(S \bigcup S_\lambda) \times \sigma \rightarrow S$, known as the transition function, where S_λ denotes the preborne state that exists before the object is created.



(a) An Atomic OSD (b) A Compsite OSD

id ::= D: data_type | I: class_type | A: class_type

Si ::= data_value relational_operator data_value

ti ::= [guard_condition] member_function_call/function calls

OSDi ::= atomic_OSD | composite_OSD

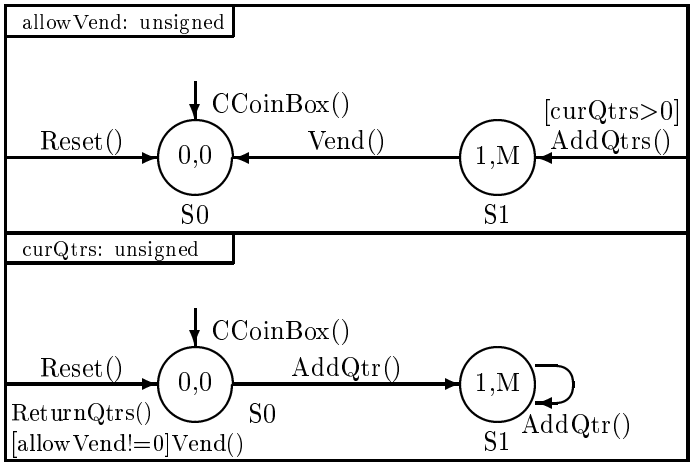
D=Data, I=Inheritance, A=Aggregation

data_type and class_type may be array or linked list.

Figure 2: Overview of the components of an OSD

A composite OSD, denoted COSD, is defined recursively as follows: 1) an aggregation of AOSD's is a COSD; 2) an aggregation of AOSD's and COSD's is a COSD.

A COSD for the erroneous coin box is shown in Figure **??**.



The semantics of a COSD can be defined formally but in this paper we choose to explain it informally. Consider the COSD shown in Figure 3, in which three concurrent AOSD's A, D, H are shown.

Suppose A is in state B, D in state F, and H in state J, denoted by (B, F, J). Now if for some reason a transition from state J to state I occurs for object H, then the next system state will be (C, G, I) by virtue

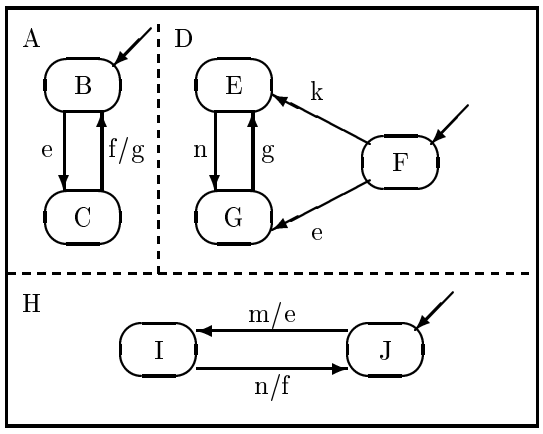


Figure 3: Interacting state machines in an OO system

of e being generated by H and triggering the two transitions in A and D. Now if transition n occurs, then the AOSD H will change state from I to J and trigger the f/g transition in A which in turn will trigger the g transition in D, resulting in (B, E, J).

5 Reversing Object State Behavior

In this section, we outline a reverse engineering method for constructing a COSD from C++ source code. In particular, we will focus on AOSD construction for a base class because a COSD can be constructed easily and recursively from AOSD's. We will present only a simplified version of the method to avoid going into details. The interested reader is referred to [27]. We will not consider pointers, pointer operations, and loops.

The constructed OSD can be used in verification and testing to detect errors in object state behavior. State testing using this method is known as program-based or white-box state testing. The other approach, which uses state machines constructed from a specification [21], is called specification-based or black-box state testing.

The method consists of three main steps: 1) symbolically executing each member function of the class; 2) identify states from the result of symbolic execution; and 3) identify transitions from the result of symbolic execution. Each of these steps will be described in greater detail in the following sections.

As our running example, we will use the coin box class presented in section 3 with the error corrected. That is, ReturnQtrs() sets both curQtrs and allowVend to zero.

5.1 A Brief Review of Symbolic Execution

The reverse engineering method derives the states and transitions from the result of symbolic execution [6] [9] [11] [12] [16]. Therefore we describe briefly the symbolic execution method in this section.

Symbolic execution is a validation technique in which a program is executed using symbolic values rather than literal values. The result of a symbolic execution can be viewed as a set of rules, each of which consists of a path condition and a final value expression of symbolic assignments. For example, a symbolic execution of the following simple function

```
void foo (int a, int b, int &x) {
   a = a * a; x = a + b;
   if (x == 0) x = 0;
   else x = 1;
}
```

results in the following two rules:

$$a * a + b == 0 \Rightarrow x = 0$$
$$a * a + b != 0 \Rightarrow x = 1$$

where a * a + b == 0 and a * a + b != 0 are path conditions, and x = 0 and x = 1 are the final value expressions. The first rule states that if a * a + b == 0, then after executing the function, x is equal to 0. The second rule can be interpreted similarly.

Problems to be overcome in symbolic execution include symbolic handling of loops, symbolic evaluation of subscripts and pointers, treatment of nonlinear inequalities in path expressions, and the large amount of detail to be handled. Since loop handling requires much space to explain, we choose not to cover it in this paper. The interested reader is referred to [27].

Our interest in symbolic execution is based on the observation that most member functions in an OO program tend to consist of only a few statements. This implies that the complexity associated with symbolic execution would not be a problem for most member functions. Therefore, symbolic execution can be used to derive the effects of these member functions. The effects of large member functions that cannot be handled by symbolic execution will be derived semi-automatically.

5.2 Symbolically Execute Each Member Function

This is the first step of the reverse engineering method. Since symbolic execution is a well-known technique, we will only present the result of symbolic execution of the member functions. Each of the member functions CCoinBox(), ReturnQtrs(), IsAllowedVend(), Reset() has only one path, denoted P0. The path condition is always true, denoted T. The final value expression is a list of the form $< variable, value >$, meaning that after symbolically executing the member function, the variable obtains the value. The member functions AddQtr() and Vend() each has two paths, denoted P0 and P1, respectively. The result of symbolic execution is summarized in Table **??**.

Path	**Path Condition**	**Final Expr**	**Return Value**
$CCoinBox().P0$	T	$< totalQtrs, 0 >$ $< allowVend, 0 >$ $< curQtrs, 0 >$	$void$
$Reset().P0$	T	$< totalQtrs, 0 >$ $< allowVend, 0 >$ $< curQtrs, 0 >$	$void$
$AddQtr().P0$	$\neg(curQtrs > 0)$	$< curQtrs, curQtrs + 1 >$	$void$
$AddQtr().P1$	$(curQtrs > 0)$	$< curQtrs, curQtrs + 1 >$ $< allowVend, 1 >$	$void$
$ReturnQtrs().P0$	T	$< curQtrs, 0 >$ $< allowVend, 0 >$	$void$
$IsAllowedVend().P0$	T		$allowVend$
$Vend().P0$	$(IsAllowedVend() == 0)$		$void$
$Vend().P1$	$\neg(IsAllowedVend() == 0)$	$< totalQtrs, totalQtrs + curQtrs >$ $< curQtrs, 0 >$ $< allowVend, 0 >$	$void$

5.3 Identification of States

This is the second step of the method. It identifies states for some of the primitive type data members. To simplify explanation we will only consider integer data types. The method can be extended to other data type without much difficulty.

The aim of creating states is to explain the behavior of the class based on the values the data member takes. The value of the data member effects the class behavior when it takes part in a decision (condition), the evaluation of which at run time controls the execution path. We will exploit this to partition the domain of the data member into intervals where the values of data member lead to different execution paths.

We need to define some terminology. An arithmetic expression is a variable expression involving one or more variables. An atomic condition is an expression involving two variable expressions connected by one of the relational operators: $>, <, ==, >=, <=$ and $!=$. Compound conditions are formed by atomic conditions and logical connectives $\wedge, \vee$, and $\neg$ as usual. A conditional literal is either an atomic condition or the negation of an atomic condition. If the only variable of a conditional literal is a data member d, then it is called a conditional literal in d. Finally, We use m to represent the min value and M the max value of the domain of any data member d.

The identification of states for a data member d is achieved by the following steps:

1. Examine all the path conditions of the member functions to look for conditional literals in d.
2. For each conditional literal identified in the last step, form intervals of the domain of d such that the conditional literal is evaluated to either TRUE or FALSE for all values in the interval. The following examples show how this can be done:

conditional literal	intervals formed
d > 8, d <= 8	[m,8], [9,M]
d < 8, d >= 8	[m,7], [8,M]
d == 8, d != 8	[m,7], [8,8], [9,M]

3. The above steps result in a set of intervals over the domain of d. Form states from the intervals as described below.
   (a) Select any two intervals A and B that intersect.
   (b) Form intervals $A - (A \cap B)$, $B - (A \cap B)$ and $A \cap B$.
   (c) Replace A and B by above intervals.
   (d) Repeat the above steps until no intersecting intervals exist.
   (e) The intervals created are the states.

Example. To illustrate the above steps we will derive the states for the data member curQtrs. We will use an abstract example to illustrate some of the steps that have not been covered.

1. From the path conditions we find that the conditional literals for the data member curQtrs are "curQtrs > 0", and "$\neg$(curQtrs > 0)".
2. The two conditional literals yield the following intervals: [m,0], [1,M], since m=0 for unsigned, we have [0,0] and [1,M].
3. Since the intervals do not overlap, they are the states.

The reader will notice that the behavior of the coin box in the vending machine is that if there is at least a quarter in the coin box and another is added then it allows vending (as far as coin collection is concerned). This is the semantics that has been captured by the states. Similarly, the states for allowVend are: [0,0] and [1,M].

Example involving intersecting intervals. If a data member d gives the following intervals: [m,10], [11,M], [m,5], [6,M] then using the steps to form states from intersecting intervals we obtain [m,5], [6,10], [11,M].

5.4 Construction of Transitions

In this step we construct the transitions between the states of a data member d. A data member changes its state if its value is changed. It is possible for a transition to start and end at the same state. We will make a simplifying assumption that the data members are defined only by the member functions of its class. That is all the data members are private. Note that any direct assignment to a data member by a non-member can be replaced by a member function that assigns a value to the data member.

A member function has one or more execution paths. A path may update a data member only if the path condition is satisfied. Thus a state transition from a pre-state to a post-state can occur if the member function contains a path whose path condition can be satisfied by the pre-state; and the final value expression of the data member can be satisfied by the post-state; that is, it lies in the interval representing the post-state.

In the following discussion we will first deal with function members that directly define a data member. The method will be extended later to deal with member functions that call another function or member function to define the data. The reader is

reminded that we have restrict ourselves to integer type and are dealing with one data member at a time. That is, the state machine constructed is an AOSD.

We will use the following notations. By definition, a state S_i is an interval [l,u]. A data member is said to be in state S_i if its value lies in the interval. We will use $S_i(x)$ to denote the expression that $(x \geq l) \wedge (x \leq u)$, where x is either a data member or a variable expression. We will use PC to denote a path condition and E the final value expression of a data member d produced by a path. Both E and PC may have function calls but their treatment is postponed to a later section to make the presentation easier to understand.

State transitions are constructed by the following steps:

1. Create a set RS of reachable states, initially set to empty.
2. Examine the states identified in the last section, add those that satisfy the final value expression produced by a constructor to RS. These states are the initial states. (If the class does not have a constructor, then add all the states to RS[1].)
3. Select a state $S_i \in$ RS as a pre-state. If S_i is an initial state, then add the constructor transition leading to S_i and label it accordingly.
4. Select a path P_k which defines the data member d. Let PC_k be the path condition of P_k and E the final value expression of d for this path. The path can cause a transition from S_i if $S_i(d) \wedge PC_k$ is consistent (i.e., satisfiable). If this is not so, reject P_k and select the next path.
5. Identify all the post states for the transitions from S_i due to the path P_k as follows:
   (a) Select a state S_j from the set of states, including the initial and potential states, of data member d identified in the last section. S_j is a post-state for a transition from S_i due to P_k if $(S_i(d) \wedge PC_k) \models S_j(E)$. That is, $S_j(E)$ must be true for all cases in which $(S_i(d) \wedge PC_k)$ is true.
   (b) If S_j satisfies the above criteria, then the transition from S_i to S_j is constructed. The guard condition for this transition is $S_i(d) \wedge PC_k \wedge S_j(E)$[2]. If the pre-state implies the guard condition, then the guard condition is omitted.
   (c) Add S_j to RS, i.e., RS = RS $\cup$ { S_j }.
6. Repeat from step 4 till there are no more paths that define the data member d.
7. Remove S_i from RS set.
8. Repeat from step 3 till RS is empty.

Example. We show how the transitions for the AOSD's for curQtrs and allowVend are constructed. We use $\delta(S_i, t) = S_j$ to denote the transition from S_i to S_j by t. Note the states S_i, S_j etc. will be denoted by their intervals, such as [0,0].

1. RS = { }.
2. Since [0,0] satisfies the final value expression of curQtrs as produced by the constructor, RS = {[0,0]}.
3. We select [0,0] $\in$ RS as a pre-state. Since [0,0] is an initial state, we add $\delta(\lambda, CCoinBox()) = [0,0]$ to the set of transitions. Recall λ denotes the object preborne state, i.e., a state before the creation of the object.
4. From the result of symbolic execution, we know the function paths that define curQtrs are: Reset().P0, AddQtr().P0, AddQtr().P1, ReturnQtrs().P0, and Vend().P1. Consider AddQtr().P0, which increments curQtrs. Its path condition, which is $\neg(curQtrs > 0) \equiv (curQtrs \leq 0)$, is consistent with the initial state, which asserts $curQtrs == 0$. Therefore, AddQtr().P0 can be applied to [0,0].
5. The execution of AddQtr().P0 increments curQtrs. That is, curQtrs == 1 after executing the path. This is true in [1,M]; and hence, [1,M] is a post-state. Thus, we have

$$\delta([0,0], [(curQtrs == 0) \wedge !(curQtrs > 0) \wedge (curQtrs + 1 \geq 1)] AddQtr()) = [1, M]$$

   which is logically equivalent to (by the absortion law $A \wedge (A \vee B) \equiv A$)

$$\delta([0,0], [(curQtrs == 0)] AddQtr()) = [1, M]$$

   Since the pre-state implies the guard condition, the guard condition is omitted, resulting in

$$\delta([0,0], AddQtr()) = [1, M]$$

[1] If the class has not constructor, then objects of that class can be initialized arbitrarily by using the compiler default constructor. Since the states partition the domain of the data member, the arbitrarily initialized value must fall into one of the states. That is, we include all the states in the set of initially reachable states.

[2] This condition is simplified as usual and is omitted in this paper.

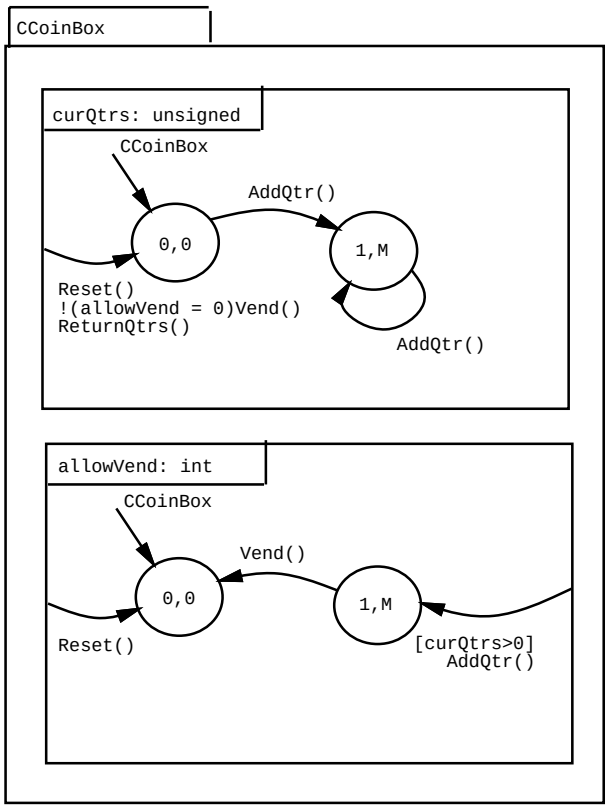


Figure 4: Composite object state diagram for class CCoinBox

6. Since [1,M] is reachable from [0,0], add [1,M] to RS; now RS = {[0,0], [1,M]}.
7. Repeating above steps constructs the following transitions:
$$\delta([0,0], Reset()) = [0,0]$$
$$\delta([0,0], ReturnQtrs()) = [0,0]$$
$$\delta([0,0], [allowVend! = 0]Vend()) = [0,0]$$
8. Removing [0,0] from RS resulting in RS = {[1,M]}.
9. Repeating the process for the state [1,M] produces the following transitions:
$$\delta([1,M], AddQtr()) = [1,M]$$
$$\delta([1,M], Reset()) = [0,0]$$
$$\delta([1,M], ReturnQtrs()) = [0,0]$$
$$\delta([1,M], [allowVend! = 0]Vend()) = [0,0]$$

Similarly, the transitions for allowVend can be constructed. In summary, the AOSD's for curQtrs and allowVend, and the COSD for the CCoinBox class are shown in Figure 4.

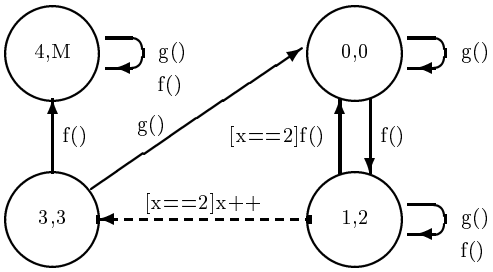
5.5 Dealing with Function Calls

Function calls are processed in our method in an incremental approach. However, it is not possible to describe all the details in this paper. Therefore, we outline in this section how to generate transitions for member functions that call other functions. We will report the details in another publication. The interested reader is referred to [27].

We observe that a member function causes some state transitions. A chain of member function invocations may result in a sequence of transitions. Thus, transitions for the calling function can be derived from the transitions of the functions called. Consider, for example, the program in Figure 5a). Suppose the transitions for g() have been constructed, as shown in Figure 5b).

```
class foo {
unsigned x;
public:
foo() { x=0; }
void g() { if (x==3) x = 0; }
void f() { x++; g(); }
};
```

a) function f() calls g()



b) constructing a transition for a calling function

Figure 5: Transitions for member functions that call other functions

The construction of f()'s transitions then can be derived. Conceptually, we can view the code segment consisting of x++ as a function (call). Suppose we want to derive a transition for f() in state [1,2]. We proceed as follows:

1. Construct a temporary transition for x++ from state [1,2]. The resulting state is either [1,2] or [3,3], depending on x<2 or x==2. We show in Figure 5b) only the transition to [3,3] in dashed directed line to indicate it is a temporary transition.
2. Since f() invokes g() after x++, we look for g() transitions going out of [3,3]. There is one from [3,3] to [0,0].
3. Since x++ changes the state from [1,2] to [3,3] and g() from [3,3] to [0,0], we infer that f() changes the state from [1,2] to [0,0] under the condition that x==2.

6 Object State Testing

In this section, we first show that the traditional function-oriented state testing cannot detect some object state errors. We then show the effectiveness of object state testing using the coin box example.

6.1 Limitation of a Traditional Approach

Traditional state testing is limited to a single state machine. In the traditional approach, a spanning tree is first generated from a state machine (denoted STD) as follows [5]:

1. Starting from the initial state S_0, the root of the test tree is constructed and labeled by S_0.
2. We now examine the nodes in the tree one by one. Let the node being examined be labeled by S_i. If S_i has already occurred at a higher level in the tree, then the node becomes a terminal node and will not be expanded. Otherwise, if there is a transition t leading from state S_i to state S_j in the STD, then we attach a branch and a successor node to S_i. The branch is labeled t and the successor node is labeled S_j.

The tree is then used to generate the test cases, each of which is a sequence of transitions starting from the root (representing the initial state) and ending at any node.

Consider the COSD in Figure **??**, which consists of two AOSD's each of which is a single state machine. The AOSD for allowVend does not have the ReturnQtrs() transition, and hence, one cannot generate the desired error detecting sequence consisting of AddQtr(); AddQtr(); ReturnQtrs(); Vend(). So we consider the AOSD for curQtrs. A spanning tree for this machine is shown in Figure 6. From the tree we cannot derive the desired test sequence AddQtr(); AddQtr(); ReturnQtrs(); Vend(), because the right branch terminates at S1 due to the loop labeled by AddQtr() in the AOSD for curQtrs. Although a flat state machine such as the one in Figure 1 can be used to detect the error, its complexity is in general high, as addressed in section 3.

6.2 Error Detection Using COSD

In our approach, test cases for testing the state behaviors of the member data is generated by using Chow's method. However, test cases for the state behavior of a class and interclass interactions are generated differently. We first explain how this is done through the abstract COSD in Figure 3. We then show how the method can be used to generate a test case to detect the error described earlier.

The nodes of a test tree for a COSD represent the composite states of the COSD. The edges of the tree represent transitions between the states. If the COSD contains k AOSD's, then each state is represented by a k-tuple, where the i-th component

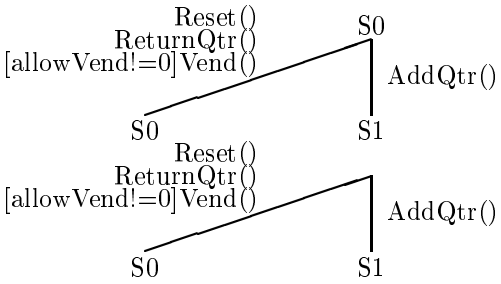


Figure 6: A spanning tree for the AOSD of curQtrs

denotes the state of the i-th AOSD. Since an AOSD may have more than one initial state and not all of them can be reachable from the other states, there could be several test trees for a COSD. Each of these test trees has a particular initial composite state. Stating more formally, a test tree for a COSD of k AOSD's is constructed as follows:

1. For each of the k AOSD's, construct a set of (inreachable) atomic initial states, denoted $AIS_i, i = 1, 2, \ldots, k$, as follows. If AOSD$_i$ has m initial states whose only incoming edge is labeled by a constructor, then, include these m initial states in AIS_i. If AIS_i is empty, then include any initial state in AIS_i.
2. Compute the set of composite initial states

$$CIS = AIS_1 \times AIS_2 \times \ldots \times AIS_k$$

   In the following discussion, each composite state is denoted $(S^1_{i1}, S^2_{i2}, \ldots, S^k_{ik})$, where the superscripts denote the AOSD's and the subscripts denote the state of an AOSD.
3. For each composite initial state $(S^1_{i1}, S^2_{i2}, \ldots, S^k_{ik})$ in CIS, construct a test tree as follows:
   (a) Starting from the initial state $(S^1_{i1}, S^2_{i2}, \ldots, S^k_{ik})$, the root of the test tree is constructed and labeled by $(S^1_{i1}, S^2_{i2}, \ldots, S^k_{ik})$.
   (b) We now examine the nodes in the tree one by one. Let the node being examined be labeled by $(S^1_{j1}, S^2_{j2}, \ldots, S^k_{jk})$.
   (c) If $(S^1_{j1}, S^2_{j2}, \ldots, S^k_{jk})$ has already occurred at a higher level in the tree, then the node becomes a terminal node and will not be expanded. That is, go to step 3b and examine the next node; otherwise, continue.
   (d) If there is a transition t leading from state S^p_{jp} to state S^p_{jq} in the AOSD$_p$, then we attach a branch and a successor node to $(S^1_{j1}, S^2_{j2}, \ldots, S^k_{jk})$, the branch is labeled t and the successor node is labeled $(S^1_{j1}, S^2_{j2}, \ldots, S^p_{jq}, \ldots, S^k_{jk})$.
   (e) If t also triggers other transitions (such as the f transition of A triggering the g transition of D in Figure 3), then the corresponding component(s) of the successor node is updated accordingly. This step is repeated until no transition can be so triggered.
   (f) Repeat from step 3b until no expansion is possible.

The tree, though may be large, must be finite, since k is finite and each AOSD has only a finite number of states.

Consider, for example, the COSD in Figure 3. In the initial state (B, F, J), three transitions, i.e., e, m, and k, can cause state changes. Thus, the initial state is expanded by three branches, labeled, respectively by e, m, and k. These transitions lead to three different states: (C, G, J), (C, G, I), and (B, E, J). These states are expanded in the same way, leading to more branches and states. A state is not expanded if it has occurred previously. This is because any state that can be reached from the current state can also be reached from the previously occurred state; and hence, there is no need to expand the current state. Figure 7 shows part of the tree constructed, where states enclosed in rectangles are not to be expanded further since it has occurred earlier. Test sequences can be generated from the partial paths in the tree to test the interacting behavior of an OO system.

Consider now the COSD in Figure **??** for the incorrect coin box program. A spanning tree for this COSD is shown in Figure 8, from which we can derive the test sequence AddQtr(); AddQtr(); ReturnQtrs(); Vend() (i.e., the second rightmost branch), which should detect the error in the implementation of the ReturnQtrs() member function.

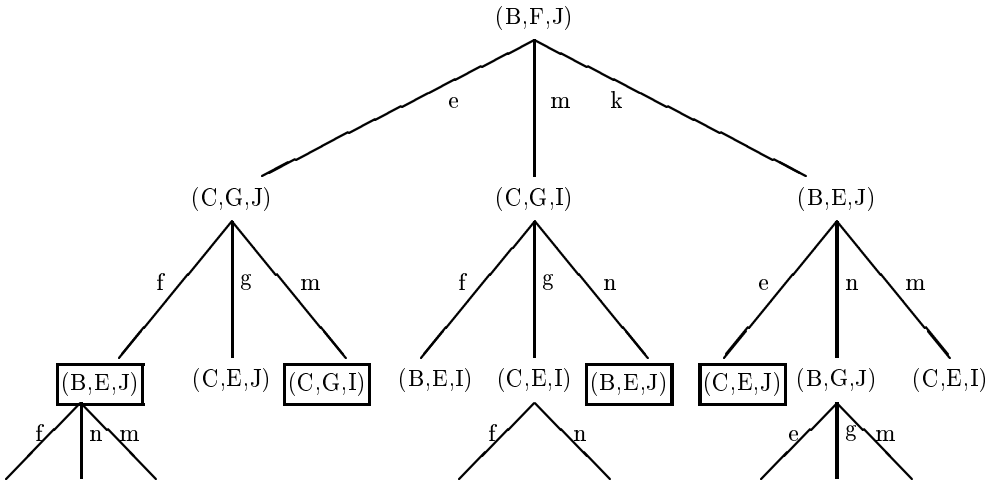


Figure 7: A partial tree presenting execution of concurrent objects

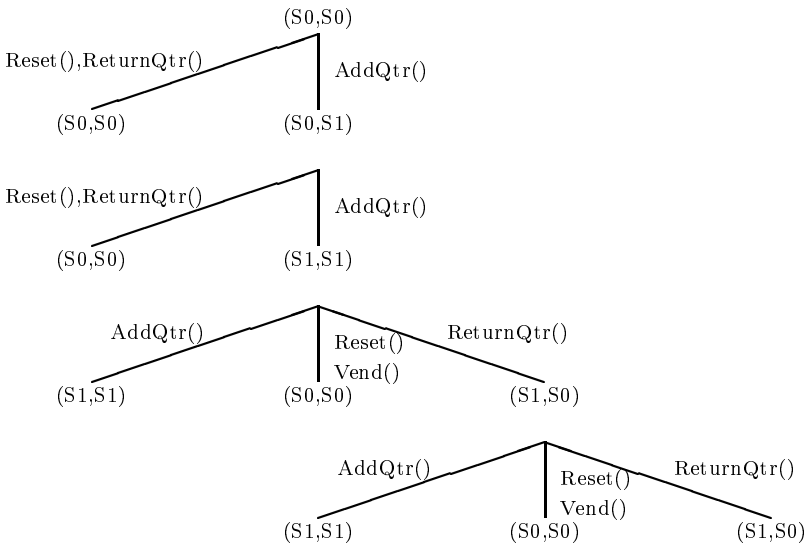


Figure 8: Test tree showing the execution sequences of a COSD

7 Conclusions and Future Work

We have presented an object state test model, which is a hierarchical, concurrent, communicating state machines. We have outlined a reverse engineering method for extracting object state behavior from C++ source code. We showed that state dependent errors are difficult (if not impossible) to detect by using conventional structural testing, functional testing, and single state machine testing methods. We illustrated that state errors could be effectively detected by the new approach reported in this paper.

The proposed method has been applied to some example programs, including an elevator application and a vending machine application. We are in the process of implementing the reverse engineering method. We plan to apply the method to large, realistic programs to further assess its effectiveness.

8 Acknowledgment

The material presented in this paper is based on work supported by the Texas Advanced Technology Program (Grant No. 003656-097), Fujitsu Network Transmission Systems, Inc., Hewlett-Packard Company, and the Software Engineering Center for Telecommunications at UTA.

9 References

[1] B. Beizer, "Software Testing Techniques," 2nd ed., Van Hostrand Reinhold, 1990.

[2] G. Booch, "Object-oriented Development," IEEE Trans. Software Eng., Vol. SE-12, No. 2, pp. 211 - 221, February 1986.

[3] Dennis de Champeaux, Al Anderson, and Ed Feldhousen, "Case study of object-oriented software development," OOPSLA'92, pp. 377–391, 1992.

[4] T. J. Cheatham and L. Mellinger, "Testing Object-Oriented Systems," in Proceedings of the 18th ACM Annual Computer Science Conference, pp. 161-165, ACM Inc., New York, 1990.

[5] T. S. Chow, "Testing software design modeled by finite- state machines," IEEE Trans. Software Eng., Vol. SE-4, No.3, May 1978. pp. 178 - 187.

[6] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Transactions on Software Engineering, Vol. SE-2, no. 3., 1976.

[7] P. Coad, "Object-Oriented Design," Englewood Cliffs, N. J.: Yourdon Press, 1991.

[8] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or how to use Statecharts in object-oriented design," IEEE Trans. on Software Engineering, vol. 18, no. 1, pp. 9 - 18, January 1992.

[9] Coward, P D, "Symbolic execution and testing," Information and software technology, Volume 33, Number 1, 53, January, 1991.

[10] Alan A. Davis, Software Requirements Analysis and Specification, Prentice-Hall, 1990.

[11] Dillon, Laura K, "Using Symbolic Execution for Verification of Ada Tasking," Programs, ACM transactions on programming languages and systems, Volume 12, Number 4, 643, October, 1990.

[12] Girgis, M. R, An experimental evaluation of a symbolic execution, system, Software engineering journal, Volume 7, Number 4, 285-290, July, 1992.

[13] D. Harel, "On visual formalisms," CACM, Vol. 31, No. 5, pp. 514 - 531, May 1988.

[14] D. Harel, et al, "STATEMATE: a working environment for teh development of complex reactive systems," IEEE Trans. on Software Eng., Vol. 16, No. 4, pp. 403 - 414, April 1990.

[15] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structure", Proc. of 14th International Conf. on Software Engineering, 1992.

[16] Kneuper, R, "Symbolic execution: a semantic approach," Science of computer programming, Volume 16, Number 3, 207, October, 1991.

[17] Naik, Kshirasagar, Sarikaya, Behcet, and "Testing communication protocols," IEEE Software 9:27-37 Jan 1992.

[18] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Design Recovery for Software Testing of Object-Oriented Programs," Proc. of the Working Conference on Reverse Engineering, pp. 202 – 211, Baltimore Maryland, May 21 – 23, IEEE Computer Society Press, 1993.

[19] D. Kung, N. Suchak, P. Hsia, J. Lin and Y. Toyoshima, "Detecting and Locating Memory Leak in Object-Oriented Programs," Proc. of 10th International Conference on Testing Computer Software, Washington D. C., June 15 – 17, 1993.

[20] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "Firewall, regression testing, and software maintenance of object oriented systems," to appear in Journal of Object Oriented Programming, 1994.

[21] D. Kung, J. Gao, P. Hsia, and J. Samuel, "A specification-based approach for object state testing," submitted for publication, Feb. 1994.

[22] Peng, Wuxu, and Puroshothaman, S, "Data flow analysis of communicating finite state machines," ACM Transactions on Programming Languages and Systems 13:399-442 Jul 1991.

[23] D. E. Perry and G. E. Kaiser, "Adequate testing and object-oriented programming," Journal of Object-Oriented Programming, Vol. 2, pp. 13 - 19, January/February 1990.

[24] J. Rumbaugh et al., "Object-Oriented Modeling and Design," Prentice-Hall, 1991.

[25] van Sinderen, M., Ferreira Pires, L., and Vissers, C A., "Protocol design and implementation using formal methods," The Computer Journal 35:478-91 Oct 1992.

[26] M. D. Smith and D. J. Robson, "Object-oriented programming — the problems of validation," Proc. IEEE Conference on Software Maintenance — 1990. pp. 272 – 281.

[27] N. Suchak, "A reverse engineering methodology for extracting object state behavior from object oriented programs," Tech. Rep. no. 9401, Software Engineering Center for Telelcommunicaitons, UTA, 1994.

[28] Fujiwara, Susumu, Bochmann, Gregor V, Khendek, Ferhat, "Test selection based on finite state models," IEEE Transactions on Software Engineering 17:591-603, Jun 1991.

[29] P. T. Ward, "The Transformation schema : an extension of the data flow diagram to represent control and timing," IEEE Trans. Software Eng., Vol. SE-12, No.pp. 198 - 210, February, 1986.

[30] N. Wilde and R. Huitt, "Issues in the maintenance of object-oriented programs," Tech. Rep. University of West Florida and Bell Communications Research, 1991.