The object-oriented (OO) paradigm is rapidly gaining acceptance in the software industry. However, the powerful features of this new paradigm also introduce a new set of OO software testing and maintenance problems. The pioneering work in identifying these new problems includes [7, 10–12, 14, 16, 18]. The problems can be summarized as: 1) the understanding problem; 2) the complex interdependency problem; 3) the object state behavior testing problem; and 4) the tool support problem. Detailed discussions of these problems will be provided later. Our industrial experience confirms these discoveries.

In an attempt to solve these problems, the Software Engineering Center for Telecommunications at the University of Texas at Arlington (UTA) and Fujitsu Network Transmission Systems, Inc., have undertaken a major effort in the past three years to develop a methodology for OO software testing and maintenance. The results we have obtained so far include: 1) an OO test model and a reverse engineering approach to recovering the designs of C++ programs, 2) a three level schema and algorithms for data flow testing of OO programs, 3) definition and identification of class firewalls and a test strategy for regression testing of C++ programs, 4) a program-based method for object state behavior testing, and 5) a scenario-based method for integration and acceptance testing. Parts of these results have been implemented in an integrated object-oriented testing and maintenance (OOTM) environment.

Currently, most software development organizations are still in the process of observing and/or making the transition to the OO paradigm; only a few have experienced the difficulties of testing and maintaining...
an OO program. Most OO methods do not address testing. One notable exception is Fayad’s Object-Engineering Technique (OET) [4], in which test cases and test procedures are defined for object classes. A recent issue of Communications raised the issue of the importance and reported several interesting results of OO testing [2]. Strictly speaking, OO testing and maintenance tools have not been seen in the commercial market; most CASE tool vendors are advocating the use of conventional testing tools to cope with OO testing problems. The objective of this article is to share our experience in the development and application of the OOTM environment. We use the well-known InterViews library as a case study throughout the article, except for experiment and object state behavior testing, where we use a vending machine example.

Basic Object-Oriented Concepts

Objects are the basic building blocks of an OO system. In the OO paradigm, the real world is viewed as consisting of objects; hence, many real-world applications can be considered OO systems. The notion of an object includes the following:

- An object models an entity or thing in the application domain. For example, a book or an employee in the real world can be modeled by an object.
- An object has a set of attribute values that define a state of the object. For example, the status attribute of a library book may have as its values ‘available’, ‘checkout’, ‘on reserve’, ‘missing’, and ‘removed’. These values may be used to determine the state of a book object at any time. Attributes are called member data in C++.
- An object has a set of operations it is capable of performing to change its attribute values, which may cause changes to attribute values of other objects. For example, filling an order in a retail company may cause the following changes: 1) the order changes its state from ‘new order’ to ‘filled order’; 2) the customer’s balance is changed to reflect the additional amount charged to the customer; and 3) the inventory level or quantities-on-hand of the merchandise is updated to reflect the amount sold to the customer. Object operations are called member functions in C++.
- An object has an identity that can be used to uniquely identify the object, or distinguish the object from similar objects. Each object has its own identity, so that even if two objects have the same attribute values, they can still be identified by using their identities. Object identity is not relevant to our discussion in this article; we include it for the sake of completeness.

An object-oriented system may be typeless, like Smalltalk, or strongly typed, like Eiffel and C++. An object class defines the type or structure of a set of objects—that is, the attributes and their types, and the operations of the objects.

Encapsulation means modeling and storing with an object the attributes and the operations the object is capable of performing. In a conventional paradigm, the modeling of these two aspects is done separately. For example, in structured analysis, the operations that can be performed on book objects are modeled using data flow diagrams and implemented by functions/procedures, while the attributes of book objects are specified in a data dictionary and implemented by data structures.

Encapsulation is closely related to the notion of information hiding, which suggests that a software module designer should try to hide or localize the internal linkage of data structures and implementation details of the procedures. Encapsulation provides an effective way to enforce information hiding, because the data aspect of an object may be made private and access to these private data can be achieved only through operations of the object. Thus, the ripple effect of change may be minimized.

An object class is a subclass of another object class if every object of the former is also an object of the latter. The latter is called a superclass of the former. Superclass corresponds to base class and subclass corresponds to derived class in C++.

Inheritance means that properties (i.e., attributes and operations) defined for an object class are automatically defined for all of its subclasses. The most promising benefit of inheritance is software reuse, which has been utilized widely by software engineers. Since properties of a superclass are automatically defined for all of its subclasses, software that implements the operations of the superclass can be reused by the subclasses. For example, a print operation for a document object class may be reused to print a technical report if the printing of a technical report requires the same actions to be performed. OO programming also allows different printing methods for different types of documents. The appropriate method will be invoked at run time, according to the type of the document object. This ability is called polymorphism and/or dynamic binding, to be explained shortly.

An object class may have more than one superclass. For example, graduate teaching assistants are a subclass of both students and employees. This is usually referred to as multiple inheritance.

Methods inherited from a superclass must be retested in the context of the subclass, because the testing using the context of the superclass may not include all the cases that may occur in the context of the subclass [15].

Polymorphism means the ability to take more than one form: An attribute may have more than one set of values, and an operation may be implemented by more than one method. A commonly used example is from computer graphics, in which different drawing methods may be implemented for a draw operation. The drawing method (e.g., draw_arc, draw_rectangle) appropriate to the object to be drawn will be executed at run time.
Dynamic binding means the method that implements an operation is unknown until run time. It is an effective mechanism to implement polymorphism. As discussed previously, an operation may have more than one implementation. The choice of which implementation to use when an operation is invoked is determined at run time according to the types, the number of arguments, and/or the function pointed to by a function pointer.

Some Object-Oriented Testing and Maintenance Problems
As discussed in the introductory section, the major OOTM problems are 1) the understanding problem, 2) the complex interdependency problem, 3) the object state behavior testing problem, and 4) the tool support problem.

The Understanding Problem
The understanding problem is introduced by the encapsulation and information-hiding features. These features result in the “localized plan,” in which several member functions from possibly several object classes are invoked to achieve an intended functionality. Often, a member function of a class in turn invokes other member functions, resulting in the so-called invocation chain of member functions [18]. To illustrate, we show in Figure 1 a hypothetical example. The rectangles represent member functions belonging to various classes, which are denoted by capital letters. The edges represent function invocations from left to right. For example, the LOAN.check_out() function invokes LOAN.check_patron(), which in turn invokes PATRON.get_category() and then PATRON.copies_chkout(). Thus, a preorder traversal through the tree in Figure 1 gives the following sequence of member function invocations, where comments are given after ‘//’:

LOAN.check_out() // check out a book by a patron.
LOAN.check_patron() // does he borrow too many books?
PATRON.get_category() // faculty, staff, and student have different check-out limits.
PATRON.copies_chkout() // how many books has the patron checked out?

BOOK.get_type() // reference book is not allowed to be checked out.
BOOK.available() // is the book available?
COPYLIST.empty() // if the book’s copylist is empty, then it is not available.

Table 1 summarizes the number of invocation chains of different lengths for the InterViews library. The result does not take into consideration overloading, polymorphism, and dynamic binding1 because these features require actual execution of the member functions. We see that there are 4,818 member function invocation chains. The longest chain involves 14 member function calls in sequence, and the majority of cases involve chains of two to nine member functions. By “a chain of member functions calling in sequence,” we mean that the first function calls the second, which calls the third, and so forth. Thus, the longest such chain in Figure 1 is 3, not 7.

The implication of the invocation chains is that a tester/maintainer has to understand sequences of member functions and the semantics of the classes prior to preparing any test cases and/or modifying the intended functionality. Since it is necessary to understand all the parts in sufficient detail before testing/modification,

---

1 When these features are considered, the invocation chains will be longer and more difficult to comprehend, because one does not know which code segment will actually be executed at run time.
this adds tremendous complexity to testing and maintenance of OO systems.

We have also conducted an experiment to find out how much time a tester would spend to test three simple, small member functions from an elevator program written in C++. The testers are students who have learned and programmed in C++. They are required to manually prepare basis path test cases and test data for testing the member functions. Although each of the member functions is very small (a total of 15 lines of code, or an average of 5 lines per member function), it took, on average, 0.95, 0.93, and 0.79 person-hours to prepare the test data, test driver, and test stub, respectively. We know that automatic test data and test driver generation is possible and that this would save about 1.88 (i.e., 0.95 + 0.93) person-hours per member function. Automatic test stub generation is not possible, because it requires understanding of the semantics of the called functions. Therefore, reducing the effort needed to construct test stubs will result in additional savings.

The Complex Dependency Problem
The dependency problem was caused by the complex relationships that exist in an OO program. These include the inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships imply that one class inevitably depends on another class. For example, the inheritance relationship implies that a derived class reuses both the data members and the function members of a base class and hence is dependent on the base class. We have applied our tools to a version of the InterViews library, which contains 122 classes, more than 400 inheritance, aggregation, and association relationships, and more than 1,000 member functions. It does not contain template class or nested class. However, the 122 classes are related to each other, forming a strongly connected cyclic graph. Another industrial library we have examined contains many template classes, some of which are derived classes of other template classes. The InterViews library is considered a small library compared to some of the class libraries that currently exist in the industry.

The complex relationships that exist in an OO program make testing and maintenance extremely difficult:

- It is very difficult to understand a given class in a large OO program if that class depends on many other classes.
- Without sufficient insight, a tester may not know where to start testing an OO library.
- It is extremely costly to construct test stubs, since the tester has to understand the called functions, possibly create and properly initialize certain objects, and write code to simulate the behaviors and effects of the called functions.
- It is impossible to predict, and equally impossible to test, all possible uses of a template class.

- It is difficult to identify and test the effect of polymorphism and dynamic binding.
- It is difficult to identify change impact in OO maintenance, since the impact may ripple throughout the OO program through the complex dependencies [12].

The State Behavior Testing Problem
Objects have states and state-dependent behaviors. That is, the effect of an operation on an object depends on the state of the object and may change the state of the object. Thus, the combined effect of the operations must be tested [3, 5, 8, 9, 11, 17]. Consider a coin box class 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 shown in Figure 2.

B
The Tool Support Problem
CASE tools to support OO testing and maintenance are still in their infancy. Many commercial tools still implement conventional testing and maintenance methods and techniques. However, we have found these methods and techniques, though applicable, are not adequate for OO programs because they do not address the OO testing problems.

Software testing is a tedious process. It requires the preparation, execution, and analysis of tens of thousands of test cases and test data sets. Using a specification-based approach, the test cases can be derived from a software specification. This requires that the specification be written in a formal specification language so that a tool can be used to derive the test cases. The tester may or may not be required to provide test data for each of the test cases, depending on the level of detail of the formal specification. According to our observation, formal methods are rarely used in practice for various reasons. A tester has to manually prepare the test cases and test data sets. Therefore, extensive tool support is important in software testing, and OO testing is no exception.

As indicated previously, changes to an OO program may ripple throughout the entire program. Manually identifying the change impact is both difficult and time-consuming. This approach is also associated with the risk of omitting certain affected parts. An alternative is to retest the entire program, but this is very costly. Therefore, tool support is also crucial in the maintenance phase. Tools may help in the identification of the changes and change impact, generation and/or reuse of test cases and test data sets, and retesting only the parts that are changed and/or affected.

An Object-Oriented Testing and Maintenance Environment
In this section, we describe the test model and the capabilities of the OOTM environment. This information may be useful for understanding our experience (described in the next section).

The Test Model and Its Capabilities
Central to the environment is a mathematically defined test model, consisting of three types of diagrams: 1) the object relation diagram (ORD), 2) the block branch diagram (BBD), and 3) the object state diagram (OSD). These diagrams are extracted from code using a reverse engineering approach [10]. The ORD, BBD, and OSD are summarized as follows:

- An ORD represents the inheritance, aggregation, association, (template class) instantiation, uses (in template class instantiation), and nested_in relationships among the object classes and template classes. The usefulness of ORD will be explained with an example later.
- A BBD represents the control structure of a member function and its interface to other member functions so that a tester will know which data is used and/or updated and which other functions are invoked by the member function. Information contained in the BBD diagrams can be used to prepare functional test cases, structural test cases, and test harnesses in member function unit testing; to derive data dependence relations across multiple functions and objects; and to display statement, branch, and path coverage information. These capabilities are similar to conventional unit testing of the member functions [1] and hence will not be discussed further in this article.
- An OSD is similar to Statechart [6] and Rum- baugh et al.’s object state diagram. It represents the state behavior of an object class. To reduce complexity, we construct a state machine for each state-dependent data member of a class. The state behavior of a class is represented as an aggregation of the data members’ state machines. If a data

```cpp
class CoinBox
{
    unsigned totalQtrs; // total quarters collected
    unsigned curQtrs; // current quarters collected
    unsigned allowVend; // 1 = vending is allowed

public:
    CoinBox() { 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 CoinBox::AddQtr()
{
    curQtrs = curQtrs + 1; // add a quarter
    if (curQtrs > 1) // if more than one quarter is collected,
        allowVend = 1; // then set allowVend
}
void CoinBox::Vend()
{
    if (isallowedVend()) // if allowVend
    {
        totalQtrs = totalQtrs + curQtrs; // update totalQtrs,
        curQtrs = 0; // curQtrs, and
        allowVend = 0; // allowVend,
    } // else no action
}

Figure 2. C++ source code for a simple coin box
```
member is a simple datum (e.g., integer, float, or string, etc.) then the member’s state machine is called an atomic OSD (AOSD). A composite OSD (COSD) is an aggregation of AOSDs and/or COSDs. This recursive definition allows us to model the inheritance and aggregation of object state behaviors in a hierarchy of state machines.

Examples

We show in this section some examples of the application of the OOTM tools. Figure 3 shows an ORD diagram, along with the class test order for a subset of the InterViews library.

The diagram in Figure 3 indicates that Scene is a part of Canvas and World is a subclass of Scene. The test order is computed according to the degree of dependencies between the classes, so that if class A is dependent on class B, then class B should be tested before class A. A more detailed description can be found in [13]. In this way, the testing of A can reuse the data structure and functionality of B, instead of having to construct test stubs to simulate the behavior of B. We will show later that this can result in tremendous savings in OO unit testing and integration testing. The test order consists of two components, the major test order and the minor test order. The classes are tested according to their major test order, followed by their minor test order. In Figure 3 the test order is indicated by x.y, where x denotes the major test order whereas y denotes the minor test order.

The test order may be used to conduct unit class testing and integration testing. In unit testing, the individual classes are tested according to the test order. That is, each of the member functions of the class is tested using structural testing and functional testing as in conventional testing. The BBD can be used to generate the basis path test cases as well as functional test cases. The state behavior of the class is then tested using the OSD and the test utilities. In integration testing, the classes are integrated and tested according to the test order, and we focus on testing the member functions’ interfaces with other classes and interactions between the state behaviors of the integrated classes.

The complex dependencies between the classes in an OO system imply that changes to one part would ripple throughout the system. The real problem is that it is difficult for a tester to remember and record the changes and for a maintainer to identify the impact of the changes. The firewall tool we have developed automatically identifies the changes and computes the affected classes. Figure 4 shows the changes and their impact on two versions of the InterViews library subset. The figure also displays the test order for retesting the affected classes. Currently the firewall tool is based on a static approach, which does not take into consideration the polymorphism and dynamic binding features. A dynamic approach is being studied and will be used to improve the accuracy of the firewall. For a description of the firewall computation method, see [12].

The firewall tool can be used in two different ways:

• Before making actual changes, a developer can use the tool to identify the impact of the planned changes and estimate the effort in terms of number of test stubs and test cases needed to retest the affected classes.
• After making the changes, a regression tester can use the tool to identify the actual changes and their impact. The tool can also compute the optimal test order to retest the affected classes.

Object state behavior testing is an important aspect of OO testing. Our experience indicates that even though each member function of a class has been structurally and functionally tested correct according to the member function’s specification, the combined effect of invoking the member functions of the class in a certain sequence may produce an incorrect result. This is due to the failure to check the side effects of the member functions in the specifications and/or code. An example is the CoinBox program we discussed earlier. We wish to point out that the CoinBox program is a simplified version of a real application program and that the error was discovered by the programmer attempting to reconstruct the state diagram for the object class. By using the OSD tool, we generated the state diagram for the CoinBox class as shown in Figure 5. The OSD construction method can be found in [11].

The COSD for the CoinBox in Figure 5 consists of two AOSD’s for its two state dependent data members. Note that totalQtrs is also a data member of the CoinBox. But since it does not affect the state behavior of CoinBox, it does not have an AOSD. The notation and semantics of our OSD are similar to those in the literature (see [6]). That is, predicates or guard conditions are enclosed in brackets, and transitions are labeled by member functions. In our model, the states are labeled by disjoint intervals of the data member, such as [0,0] and [1,M], where M denotes the maximum value of the data member. A directed edge from the boundary of the enclosing rectangle to a state is used to represent transitions that can initiate from any state and result in the state pointed to by the directed edge. For example, the Reset() member function can be applied in any state and will result in state [0,0].

Our OSD can be characterized as hierarchical, concurrent, communicating state machines. Hierarchical because we want to support the inheritance
and aggregation features of OO programming. That is, a derived/aggregate class inherits the state machines of its base classes/component classes. Concurrent because objects are viewed as concurrent processes. And communicating because objects can send and receive messages from each other.

In the following discussion, we show how test cases can be generated from the COSD in Figure 5 to detect the error we mentioned earlier. The basic idea is to construct a test tree, as in Figure 6, from which test cases are derived.

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 composite states. If the COSD contains \( n \) AOSDs, then each state is represented by an \( n \)-tuple, where the \( i \)-th component denotes the state of the \( i \)-th AOSD. The root of the test tree is denoted by the initial states of the component AOSDs. For example, in Figure 6, the root is denoted by \((S0, S0)\). A transition from a composite state to another composite state can occur if and only if a corresponding transition can occur in an AOSD. For example, for the \( \text{curQtrs} \) AOSD, there is a transition from \( S0 \) to \( S1 \); therefore, we add the edge \( \text{AddQtr()} \) and the successor node \((S0, S1)\) to the root node in Figure 6. A directed arc from the contour of the AOSD-enclosing rectangle to a state means that there is a transition from every state to the state pointed to by the directed arc. For example, the directed arc labeled by \( \text{\{curQtrs > 0\} AddQtrs()} \) in the \( \text{allowVend} \) AOSD represents the fact that there is a transition from \( S0 \) to \( S1 \) and a transition from \( S1 \) to \( S1 \). This explains why there is a transition from \((S0, S1)\) to \((S1, S1)\) in the tree in Figure 6.

From the test tree, we can derive the test sequence \( \text{AddQtr(); AddQtr(); ReturnQtrs(); Vend()} \) (i.e., the second rightmost branch), which will detect the error in the implementation of the member functions. We have also applied the OSD tool to a traffic light example, which was used in the literature to illustrate error detection for safety critical software. The tool easily detected an error in the original code that caused a traffic light in one direction to change from green to red without the yellow warning light.

**Experience and Limitations**

We have presented OO testing problems and some of the tools we have developed to solve these problems. In particular, the test model (ORD, BBD, and OSD) makes different levels of abstraction of an OO software system to facilitate understanding (problem 1). The OSD facilitates testing of object state behavior (problem 3). The tool set generates, among other things, the test model, test order and firewall (using dependencies), and object state behavior test cases and test data (problems 2 and 4). In this section, we present our experience in the development and application of the OOTM environment.

**An OO Test Model Is Extremely Useful**

The first lesson we learned is that an OO test model is extremely useful, especially a formally defined test model. Several years ago, some of us were involved in the development of a commercial product using C++.
We were astonished by the lack of systematic OO testing methods and supporting tools to help a tester to carry out the testing process. We discovered that the traditional testing techniques and methods were inadequate, since they did not address the complexity of an OO program. When we started to test one class, we soon found out that one had to trace and understand many other classes in order to construct the test cases and test stubs. We realized that an OO test model that would provide a high level abstraction of the OO program was needed. This model should help a tester in understanding the complex relations and coping with the complex dependencies among the various components of the OO program. The development of the ORD and the test order tool was motivated by this experience. The usefulness of the OO test model is summarized as follows:

- It helps the tester and maintainer understand the structures of and relations between the components of an OO program.
- It provides the tester and maintainer a systematic method to perform OOTM. In particular, it assists the tester and maintainer to find better test strategies to reduce testing and maintenance effort.
- It facilitates the definitions and analysis of OO testing criteria.
- It facilitates the development of the various algorithms and capabilities for OOTM.

Optimal Test Order Significantly Saves Test Costs
The significance of the optimal test order is reaffirmed by the number of test stubs needed for unit testing of the InterViews library using a randomly generated test order, as shown in Figure 7. The total number of stubs required is 400 (i.e., 3.27 per class X 122 classes). If we use the result obtained in our experiment mentioned earlier, each stub requires 0.79 person-hours to prepare; then 400 stubs would require 316 person-hours, or almost 8 person-weeks. In comparison, if the optimal test order is used, then it requires only 8 test stubs. That is, a 93% saving, in terms of the number of stubs needed, is achieved. Using the test order still require 8 test stubs because there are cyclic dependencies among the classes. The weakest dependency in a cycle must be simulated by a test stub(s) so that the remaining dependencies become acyclic and a (topological) test order can then be defined for the classes. The effort or savings may be estimated more accurately by counting the number of lines of executable source code of the functions that need to be simulated by the test stubs. This approach assumes that the effort required to understand a function and construct a stub for it depends on the size of the function. A simpler approach is to assume that on average, each stub requires a certain degree of effort. The latter approach was used in the preceding material.

It should be noted that the test order assumes bottom-up testing, which is also advocated by other researchers [7, 8]. Effort is saved by significantly reducing the number of test stubs. The effectiveness of the test case, which also affects the test effort, is not considered. We believe that the test order and the test case effectiveness are two orthogonal issues and should be addressed separately. The latter depends on the test case generation technique.

Class Firewall is a Useful Concept in OO Regression Testing
Programs undergo continual changes, and OO programs are no exception. In structured programming, a programmer can strive to reduce the ripple effect and achieve a better design by enforcing the scope of effect within the scope of control. However, this design principle is sometimes difficult to enforce due to the nature of OO programming, because a decision of one object may affect the action of another object. Therefore,
without tool support, it is almost impossible to anticipate and identify the effect of change. The application of the firewall tool to the InterViews library has produced promising results. Figure 8 shows the number of affected classes due to changes to one through 10 classes.

The first case in Figure 8 is an average over 20 cases, in each of which a single class is changed. We see that less than 50% of the classes need to be retested. The next two cases concern the deletion of one or two randomly selected classes. Most of the cases show that fewer than 75% of the classes need to be retested. The average is 15.05 affected classes per class changed. In other words, only 12.33% of the 122 classes need retesting, achieving a 87.67% savings in terms of the number of classes need to be retested. It should be noted that retesting effort may be affected by the effectiveness of the test cases. Generally speaking, the more effective the test cases, the less retest effort is required with respect to a given quality requirement. This issue is orthogonal to the firewall concept. In other words, for any given set of classes, effort can be saved by using more effective test cases.

We anticipate that the tool will be particularly useful in the maintenance phase. Without using this tool, one has to document the changes and identify the effect according to one’s knowledge of the system. This is both time-consuming and inaccurate. With this tool, the problem is solved in seconds.

Using ORD and Test Order in Integration Testing
Integration testing is not an easy task. Traditionally, there are several integration strategies, including top-down, bottom-up, and sandwich approaches [1]. It has been recognized that a bottom-up strategy is preferred for OO programs. However, how to conduct bottom-up OO testing is still an open problem. One notable contribution is [7], in which a bottom-up methodology was proposed for testing the inheritance hierarchy of OO programs. Jorgensen and Erickson [8] proposed and outlined a five-level testing approach that also advocates bottom-up integration. Our experience indicates that the ORD and the test order not only are useful for conducting class unit testing but also provide a detailed road map for conducting integration testing. That is, after class unit testing is conducted, the classes are to be integrated according to the test order. In this way, the effort required to construct test stubs and test drivers can be reduced to a minimum.

Consider, for example, the integration testing of the InterViews library. Figure 9 shows the statistics of test stubs required for 100 random integration sequences. We see that the average number of stubs required is 191.88 per sequence. If each stub requires 0.79 person-hours to prepare, then testing would require about 152 person-hours or almost 4 person-weeks. When the test order is used, only 8 test stubs are needed.

Limitations and Enhancements
One obvious limitation is that we focus only on white-box testing; hence the limitations of white-box

Figure 8. Statistics for change impact identification

Total no. of classes changed: 57; total affected: 857.76; average affected: 15.05
* Average over 20 cases; a number of classes are changed.
** One randomly selected class is deleted.
*** Two randomly selected classes are deleted.
**** A number of randomly selected classes are changed.

Fewer test stubs are required than the random unit test sequence because the result in Figure 9 is an average over 100 sequences and the result in Figure 7 is for one random sequence, which may happen to be a costly one.
testing are the limitations of our OOTM environment. Although the BBD interface can be used to generate functional testing cases and test data, the testing of member functions is only a small portion of OO testing.

Applying the ORD tool to realistic programs has revealed some design and implementation limitations of our OOTM environment. The ORD display tool works fine for a small set of classes. The display would spread over several screens/paper sheets when the number of classes is large. For example, the display of the complete InterViews library takes up five screens, and a user has to use the scroll bar to view the entire ORD diagram. This “scaling-up” problem is common to many CASE tools. Three promising solutions have been proposed:

- Supporting multiple class libraries. A large OO system involves several libraries. A library dependency graph, in which nodes represent libraries and directed edges represent interlibrary dependencies, can be created to provide an additional level of abstraction. A user can click on a node to view a library’s ORD or on an edge to view the relationships among the classes of two libraries. Interlibrary dependencies are identified from “#include” statements.
- Supporting multiple subsystems. Another level of abstraction can be provided by a subsystem graph, in which nodes represent subsystems and directed edges represent interactions among the subsystems. A user can click on a node to view a subsystem’s ORD or on an edge to view the class relationships among the classes of two subsystems. A subsystem may contain a main() function, and it and its related files may be assumed to reside in a header file directory and a source file directory, respectively.
- Supporting large ORD display. Several techniques can be used to effectively display a large ORD in a library or subsystem. For example: 1) A matrix representation, in which rows and columns denote classes and entries denote relationships among the classes, is a straightforward alternative. 2) A diagonal matrix is a matrix in which the nonblank entries are clustered around the diagonal. A user can navigate along the diagonal and view the clusters one at a time. 3) The ORDs associated with clusters can be displayed one at a time. 4) An ORD that contains one or more types of relationships can be displayed. 5) A user may specify a class and a distance so that only classes that are within this distance, measured in terms of the length of the path from the specified class, will be displayed. 6) A user may request display of only the classes that contain a particular character string.

Our OSD construction is based on symbolic execution to identify the effects of the member functions of a class. Loop statement handling is a known weakness of symbolic execution. Our solution is to symbolically execute loops using boundary values and include the boundary values in the guard conditions of the transitions. For example, if the values of a loop variable \( x \) can range from \( a \) to \( b \), then as boundary values, we use \( x = a - 1, x = a, x = a + 1, x > a + 1 \) and \( x < b - 1, x = b - 1, x = b, x = b + 1 \), respectively, to execute the loop and derive the effect. We are satisfied with the results for testing purposes. However, since only boundary values are used, the OSD constructed may not be complete.

An important lesson we have learned is that the complexity of object state behavior testing is very high, despite our use of an efficient object state model. When we applied the tool to a class of a vending machine simulator, the AOSD of a data member (i.e., CurrentCash) consists
of 7 states and 94 transitions. The number of state transition sequences is 881. This implies that 881 test cases and test data suites should be generated and exercised. Without tool support, a tester would have to spend a lot of time to prepare the test cases, and test data suites, and execute the tests.

However, even with tool support, the tester still had to specify the expected result for each of the 881 test cases. OO systems usually involve many classes. Class libraries containing thousands or even tens of thousands of classes are not uncommon. Therefore, tools to further reduce the tester’s effort are needed.

Objects interact through message passing, resulting in concurrent state transitions in several state machines. The combined state space of these state machines can be unmanageably large. This makes test case generation and test execution extremely difficult, time-consuming, and costly. Our experience in testing interacting object state behavior is still very limited. We had not tried large examples due to the complexity and lack of tool support. (The tool is under development.) The good news is that it seems reasonable to test only those events that cause concurrent state transitions, because the nonconcurrent state transitions have been tested in AOSD testing prior to COSD testing of object interactions.

We wish to point out that not all classes of an OO system require state behavior testing. Our experience indicates that, depending on the nature of the application, only a certain percentage of classes have state-dependent behaviors. We also felt that not all of the 881 transition sequences generated by the tool must be tested to ensure quality. For example, some of the sequences are subsequences of other sequences. There are other application-dependent reasons to exclude some of the sequences. We need better methods to handle the complexity of state behavior testing of real application programs.

One effective way to reduce state behavior testing effort is constraint-directed testing. A constraint specifies that under certain conditions (i.e., combination of object states) something good must eventually happen, or something bad must never happen. Thus, state behavior testing can focus on ensuring that such constraints are not violated. Effort is reduced by not testing the other transition sequences.

Conclusions and Future Directions

We have described some major OO testing problems and the OOTM environment. We also reported our experience in the development and application of the OOTM environment. Although the system is still in its prototyping stage, the tools have been ported to some companies and several other companies have expressed interest in experimenting with the system.

We are currently working on testing and regression testing of dynamic binding features. We found that the OO test model is useful for solving these problems. Another effort is the design and implementation of an OO database to support the environment.

Future research and development will focus on improving the robustness of the existing tools; enhancing the OOTM environment, including black-box testing, distributed real-time embedded software system testing, and interoperability; conducting controlled experiments on large OO programs to quantitatively assess the effectiveness of the results; and incrementally commercializing the environment for technology transfer.

Acknowledgments
The authors wish to thank Lan Chou, Premal Desai, Wen Feng, Kyung Ouk Lee, and Hwan Wook Sohn for collecting the statistics of the InterViews library. The material presented in this article is based on work supported by the Texas Advanced Technology Program (Grant No. 003656-097), the Electronic Telecommunications Research Institute (ETRI) of Korea, Fujitsu Network Transmission Systems, Inc., Hewlett-Packard Company, IBM, and SECT.

References


About the Authors:

DAVID KUNG is an associate professor in the Department of Computer Science and Engineering at the University of Texas at Arlington. Current research interests include object-oriented software testing and conceptual modeling. Author’s Present Address: Department of Computer Science and Engineering, The University of Texas at Arlington, P.O. Box 19015, Arlington, TX 76019. email: kung@homer.uta.edu

PEI HSIA is the director of the Software Engineering Center for Telecommunications at the University of Texas at Arlington, and is currently is serving as a visiting professor at the U.S. Military Academy at West Point. Current research interests include functional testing of object-oriented programs. Author’s Present Address: Department of EE and CS, USMA, West Point, NY 10996. email: hsia@eccs1.eecs.usma.edu

JERRY ZEYU GAO is a member of the technical staff at Fujitsu Network Transmission Systems, Inc. Current research interests include methodologies and supporting tools for object-oriented software testing and maintenance, software testing techniques, and requirements engineering.

YASUFUMI TOYOSHIMA is a vice president of Fujitsu Network Transmission Systems. Current research interests include software engineering for object-oriented technology and software process management.

CHRIS CHEN is a senior manager of Fujitsu Network Transmission Systems. Current research interests include software engineering methods and object-oriented technology and test techniques.

Authors’ Present Address: Fujitsu Network Transmission Systems, Inc., 2540 N. First St. Suite 201, San Jose, CA 95131.

YOUNG-SI KIM is a principal member of engineering staff working in the Switching Technology Division of the Electronics and Telecommunications Research Institute. Current research interests are switching system software, distributed real-time system software, software testing, and object-oriented software development. email: yskim@tdx.etri.re.kr

YOUNG-KEE SONG is a principal member of the engineering staff working in the Switching Technology Division of the Electronics and Telecommunications Research Institute. Current research interests include software development environments, CASE tools, software testing, and object-oriented software development.

Authors’ Present Address: Switching Technology Division, ETRI, P.O. Box 106, Yousong, Taejon, Korea. email: yksong@tdx.etri.re.kr

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/95/1000 $3.50