On Regression Testing of Object-Oriented Programs

David C. Kung, Jerry Gao, and Pei Hsia
Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, Texas

Yasufumi Toyoshima and Cris Chen
Fujitsu Network Transmission Systems, Inc., San Jose, California

Regression testing is an important activity in software maintenance. Although a number of studies have addressed the problems and solutions in regression testing of traditional programs, no studies have focused on the issues and solutions in regression testing of object-oriented programs. In this article, we discuss various types of code changes of classes in an object-oriented program and present a method for identifying these changes and the affected classes. An algorithm for generating a desirable order to test the affected classes is also described. The basic model we use is an object relation graph, which depicts the inheritance, aggregation, and association relations that exist in the object-oriented program to be maintained. The test order generation algorithm can be applied to acyclic as well as cyclic object relation graphs. The results of this work have been implemented and applied to testing of many example applications, including the InterViews library.

1. INTRODUCTION

Regression testing involves retesting part of a software system after it is modified. The modification may be caused by specification or code change. The objective of regression testing is to ensure that the modified program still satisfies its requirements. To save effort and time, regression testing need only retest those parts that are affected by the modification.

Regression testing has to address four fundamental problems:

1. How to automatically identify the affected components due to changes of some components
2. What strategy should be used to retest these affected components
3. What are the coverage criteria for retesting these components
4. How to select, reuse, and modify the existing test cases (and generate new ones).

Solutions to these problems for traditional programs have been proposed during the last two decades (e.g., Fischer et al. 1981; Harrold and Soffa 1988, 1989; Laski and Szemer 1992; Leung and White, 1990; Prather and Myers, 1987; White and Leung, 1992). However, testing of object-oriented (OO) programs has received very little attention; regression testing of OO programs has received almost none.

The OO paradigm for software development introduces a number of new concepts, such as class, inheritance, encapsulation, dynamic binding, and polymorphism. These new concepts result in complex relationships between classes and their attributes. They not only introduce new testing problems as recognized in Harrold et al. (1992), Perry and Kaiser (1990), Smith and Robson (1990), and Wild and Huitt (1991), but also raise a new and challenging question of how to conduct regression testing for OO programs.

Although the existing results can be applied to regression testing of member functions of a class at the unit and integration levels, they are not suitable for testing components at higher levels, such as a
class, a group of classes, or class libraries. First, traditional approaches do not address the complex relationships and dependencies, such as inheritance, aggregation, and association, that exist between classes. Second, most traditional approaches are based on the control flow model, but class objects have state-dependent behavior that can change in various ways; hence, traditional approaches cannot be applied to class testing. Third, traditional approaches use test stubs to simulate the modules that are invoked, but in OO programs this is difficult and costly because it requires understanding of many related classes, their member functions, and how the member functions are invoked (Smith and Robson (1990); Wilde and Huit, (1992)).

In this section, we briefly review existing work on regression testing, discuss problems, and relate our work to the existing work. General discussions on regression testing can be found in Leung and White (1989) and Hartman and Robson (1988). Existing solutions to the four fundamental problems described in Section 1 are discussed in the following paragraphs.

The first problem is automatic identification of affected modules or parts. Harrold and Soffa (1988) introduced a technique for analyzing the change effects within a module. The idea is use of a data flow graph to identify the affected definition-use pairs and/or subpaths. The advantage of this approach is that test effort is reduced by retesting only the affected define-use paths and new paths. The technique has been extended so that it can also be used to identify affected procedures at the interprocedural level (Harrold and Soffa, 1989). A number of researchers have proposed different methods based on a control flow graph of a procedure/function to identify the affected control paths in a module (Laski and Szermer, 1992; Prather and Meyers, 1987).

At the module integration level, Leung and White (1990a) introduced the firewall concept to enclose the affected modules due to a module modification. The notion of a control-related firewall is defined based on a call graph. Effort is reduced by retesting only the affected define-use paths and new paths. The technique has been extended so that it can also be used to identify affected procedures at the interprocedural level (Harrold and Soffa, 1989). A number of researchers have proposed different methods based on a control flow graph of a procedure/function to identify the affected control paths in a module (Laski and Szermer, 1992; Prather and Meyers, 1987).

The second problem of regression testing is finding a cost-effective test sequence for conducting retests so that test effort and costs are minimized. The well-known test strategies include the top-down, bottom-up, and sandwich approaches (Beizer, 1990). These approaches rely on the tester to make the selection. Prather and Myers (1987) proposed an adaptive-path prefix software-testing strategy that used previous test paths as a guide in the selection of subsequent paths. Their method ensures branch coverage and consumes fewer computational resources. Harrold et al. (1992) presented an incre-
mental OO testing methodology based on class inheritance hierarchy. The approach suggests that the base class should be tested before derived classes so that the test cases and relevant information of the base class can be reused in testing the derived classes.

The third problem of regression testing concerns coverage criteria. Fischer (1977; Fischer et al. 1981) and Prather and Myers (1987), respectively, described the various retest criteria relating to path coverage of a function/procedure. Leung and White (1990a; White and Leung, 1992) used firewalls as a retesting criteria at module level to ensure that all affected modules and links between modules will be retested.

The fourth problem relates to the selection, reuse, and modification of existing test cases for retesting. Fischer (1977, Fischer et al., 1981) discussed the test case selection problem in a form of a set covering problem. The basic idea is using the concept of 0-1 integer-programming models to find the minimum test cases, which cover one of the path criteria in unit regression testing. Lee and He, (1990) also used the 0-1 integer-programming model on a test matrix to minimize test efforts in functional regression testing. Leung 1991 and White proposed a retest strategy for performing corrective regression testing. The main idea is to view regression testing as composed of two subproblems: the test selection problem and the test plan update problem. Thus, the retesting process is divided into two phases: test classification and test plan update. After the existing test cases are classified into reusable tests, obsolete tests, and retestable tests in the test classification phase, only retestable test cases and new test cases are considered as tests in the new regression test plan.

The existing methods can be applied to regression testing of member functions in classes (Fielder, 1989). Traditionally, testing uses test stubs and drivers to simulate the called functions and calling functions. In OO testing, this is both costly and difficult because the tester has to understand a chain of member functions and classes before he can construct a stub or driver.

In this article, we only address the first two problems of regression testing in OO programs: identification of code changes and their affected classes, and finding a cost-effective class retesting strategy and its implementation algorithm. The last two will be addressed in future publications. Our solution to identifying affected classes has been influenced by Leung and White's (1990a) firewall concept for modules. However, our approach aims at identifying the affected classes instead of modules. This problem has not been addressed in the literature, and its complexity, as discussed in Section 1, calls for innovative solutions. As a testing strategy, we propose to find a desirable order, called class test order, to test the affected classes. The difference between our test order and that described in Harrold et al. (1992) is that we take into consideration not only the inheritance relationship, but also the aggregation and association relationships. The notation of class test order and its associated algorithms as presented in this paper can be applied not only to class inheritance hierarchies but also to class libraries and application programs.

3. THE REGRESSION TEST MODEL

Our regression test model was originally developed for capturing and representing the complex relationships and interdependencies between the various parts of a C++ program. The model consists of three types of diagrams, that is object relation diagrams (ORD), block branch diagrams (BBD), and object state diagrams (OSD). An ORD facilitates the understanding of the inheritance, aggregation, and association relationships between classes and their dependencies. A BBD is used to facilitate the understanding of the interface and control structure of a member function in a class as well as its relationships to other data items (such as global data, class data) and function members of classes. An OSD is designed to capture the dynamic behavior of a class object, including object states (or substates) and their transitions. Although an OSD can be used to conduct the behavior-based regression testing, it is not closely related to the work presented here. Hence, it is omitted. The interested reader is referred elsewhere. (Kung et al., 1994; Kung, C., Gao, J., Hsia, P., Toyoshima, Y., and Chen, Co., 1991).

3.1 Object Relation Diagram

An ORD is introduced to represent the inheritance, aggregation, and association relationships between classes. It is used to identify the affected classes (when one or more classes are changed) and generate a cost-effective test order for testing the affected classes. Before we describe an ORD, we first briefly review the inheritance, aggregation, and association concepts used in OO modeling (Rumbaugh et al., 1991). In object-oriented programs, there are three different relationships between classes. They are inheritance, aggregation, and association. In OO programming languages (e.g., C++), an inheritance feature is provided to support the generaliza-
tion and specialization concepts and encourage code reuse in the implementation. An inheritance relation between two classes means the properties defined for an object class are automatically defined for all of its subclasses (unless selective and/or overriding inheritance are specified). Aggregation, an abstract concept, is also supported in OO programming languages through encapsulation and class features. Using the aggregation concept, a composite object can be defined based on its component objects. We call the composite object an aggregated class object. The relation between an aggregated object class and its component object class is called an aggregation relationship. An associated relationship means that two independent object classes\(^1\) associate with each other in some manner. The associations include data dependence, control dependence, or message passing between two independent object classes.

**Notion of ORD.** An ORD is defined below to capture the relationships between different classes and their objects.

**Definition 1.** An edge labeled digraph \(G = (V, L, E)\) is a directed graph, where \(V = \{V_1, \ldots, V_n\}\) is a finite set of nodes, \(L = \{L_1, \ldots, L_k\}\) is a finite set of labels, and \(E \subseteq V \times V \times L\) is the set of labeled edges.

**Definition 2.** The ORD for an OO program \(P\) is an edge-labeled directed graph (digraph) \(ORD = (V, L, E)\), where \(V\) is the set of nodea representing the object classes in \(P\), \(L = \{I, Ag, As\}\) is the set of edge labels, and \(E = E_i \cup E_{AG} \cup E_{AS}\) is the set of edges defined below.

**Definition 3.** \(E_i \subseteq V \times V \times L\) is the set of directed edges representing the inheritance relation between the classes. For any two classes \(C_1, C_2 \in V, (C_1, C_2, I) \in E_i\) indicates that \(C_2\) is a derived class of \(C_1\). In C++, \(\langle C_1, C_2, I\rangle \in E_i\) is identified if and only if one of the following declarations appears in the header files of \(P\):

- "class \(C_1\); \(C_2\)"
- "class \(C_1\); private \(C_2\)"
- "class \(C_1\); public \(C_2\)"
- "class \(C_1\); protected \(C_2\)"

**Definition 4.** \(E_{AG} \subseteq V \times V \times L\) is the set of directed edges representing the aggregation relation between the classes. For any two classes \(C_1, C_2 \in V, (C_1, C_2, Ag) \in E_{AG}\) indicates that class \(C_1\) contains one or more objects of class \(C_2\). In C++, \((C_1, C_2, Ag) \in E_{AG}\) is identified if and only if one of the following conditions holds:

- Objects of class \(C_2\) are declared as the attributes of class \(C_1\). This is called automatic aggregation.
- Objects of class \(C_2\) are declared as the static attributes of class \(C_1\). This is called static aggregation.
- Objects of class \(C_2\) are dynamically created by the member functions of class \(C_1\). This is called dynamic aggregation.

For any given C++ program \(P\), they can be identified from C++ source code. Some examples follow:

**Identifying automatic aggregation.**

```cpp
class C1 {
    ... 
    C2 b; // an instance of C2 is part of C1
    C2[m]; // an array of instances of C2 is part of C1
    ...
};
```

**Identifying static aggregation.**

```cpp
class C1 {
    ... 
    static C2 b; // a static instance of C2 is part of C1
    static C2[m]; // a static array of instances of C2 is part of C1
    ...
};
```

**Identifying dynamic aggregation.**

```cpp
class C1 {
    ... 
    C2 b; // a dynamically created instance of C2 is part of C1
    C2[m]; // an array of dynamically created instances of C2 is part of C1
    C1(); // constructor for C1
    ...};
```

**Definition 5.** \(E_{AS} \subseteq V \times V \times L\) is the set of directed edges representing the association relation between two independent classes.\(^2\) For any two independent classes \(C_1, C_2 \in V, \langle C_1, C_2 As\rangle \in E_{AS}\)

---

\(^1\) Two object classes are independent if there are no aggregation and inheritance relations between them.

\(^2\) There are no inheritance as well as aggregation relationships between two classes.
indicates that class $C_1$ associates with class $C_2$ in the following three ways:

- Class $C_1$ uses data members of class $C_2$. This is called data dependence.
- Class $C_2$'s member functions are invoked by some member functions of class $C_1$. This is called message passing.
- Class $C_2$'s objects are defined as formal parameters of member functions in class $C_1$. This is called object parameter passing.

For any C++ source program $P$, $(C_1, C_2, A) \in E_{AS}$ is identified if one of the above conditions are satisfied. Some examples follow:

- Class $C_1$ depends on one or more data members of class $C_2$.
  (1) $C_1::f(\ldots)$
  \[
  \ldots \mathrm{if} \ (\mathrm{b.}\mathrm{d} \ \text{relational\_operator} \ \text{data-value}) \ \{
  \quad \mathrm{//} \ \mathrm{b} \ \mathrm{is} \ \mathrm{an} \ \mathrm{object} \ \mathrm{of} \ C_2, \ \mathrm{d} \ \mathrm{is} \ \mathrm{a} \ \mathrm{data}
  \quad \mathrm{member} \ \mathrm{of} \ b.
  \quad \mathrm{//} \ \mathrm{b} \ \mathrm{has} \ \mathrm{been} \ \mathrm{defined} \ \mathrm{or} \ \mathrm{created} \ \mathrm{outside} \ \mathrm{the}
  \quad \mathrm{scope} \ \mathrm{of} \ C_1.
  \quad \ldots \}\]
  \[
  \ldots \mathrm{f}(\ldots C_2::b, \ldots); \quad \mathrm{//} \ b \ \mathrm{has} \ \mathrm{been} \ \mathrm{defined}
  \quad \mathrm{outside} \ \mathrm{the} \ \mathrm{scope} \ \mathrm{of} \ C_1 \ \mathrm{and}
  \quad \mathrm{//} \ \mathrm{passed} \ \mathrm{as} \ \mathrm{a} \ \mathrm{parameter} \ \mathrm{to} \ C_1.
  \ldots \]

As an example, Figure 1 depicts an ORD that represents the relationship between the classes Car, Vehicle, Engine, Tire, and Person.

### 3.2 Block Branch Diagram

A BBD facilitates the understanding of the member functions and their relationships to the global data, class data, and other member functions. Figure 2 shows the basic components in a BBD. These components are explained as follows:

- The large block displays the BBD body, denoted $B$; it encapsulates the program graph for the member function.$^3$
- The upper left block displays the global and class data that are used by the member function; this is denoted by $D_u$.
- The upper right block displays the input/output parameters, denoted $P$, of the member function.
- The bottom left block displays the global and class data that are defined (i.e., updated) by this member function; this is denoted by $D_d$.
- The bottom right block displays functions that are called by this member function; this is denoted by $F_c$.

Formally, the BBD for a member function $f$ is a quintuple

$$BBD_f = (D_u, D_d, P, F_c, B)$$

where the components are as defined above. When no confusion can arise, we omit the subscript $f$ from $BBD_f$.

A DDD body is formally defined by a directed graph $B = (V, E)$, where $V$ denotes the set of program graph vertices and $E \subset V \times V$ the directed edges representing the control flows. As usual, $B$ satisfies the following conditions:

- There is exactly one starting vertex (which has in-degree zero) and one final vertex (which has out-degree zero).

$^3$ The program graph can be used, among others, to generate basis path test cases and test data (Beizer, 1990). However, it is beyond the scope of this article to explore this issue.
All the other vertices have in-degree one and out-degree either one or two. Except for the starting and final vertices, each vertex also satisfies the following conditions: if it has out-degree one, then it represents either a function call or a sequence of simple statements; if it has out-degree two, then it represents a decision vertex for a simple condition. Every vertex of \( B \) occurs on some path from the starting vertex to the final vertex. More detailed examples and applications of an BBD can be found in Kung et al. (1993).

4. CHANGE IDENTIFICATION

Before regression testing, one important task is to identify changes and their effects. It is very difficult to keep track of the changes when a software system is modified extensively by several persons. Thus, automatic change analysis and impact identification becomes an important capability when the modifications are performed by one group of people and regression testing is performed by another. In this section, we first discuss the different types of code changes, and then illustrate how to identify these changes using the regression test model.

4.1 Types of Code Changes

Table 1 provides a classification of code changes in an OO class library. These change types are explained as follows:

- **Data change.** Any datum (i.e., a global variable, local variable, or class data member) can be changed by updating its definition, declaration, access scope, access mode, and initialization. In addition, adding new data and/or deleting existing data are also considered as data changes.
- **Method change.** A member function can be changed in various ways. Here we classify them into three types: component, interface, and control structure changes. Component changes include adding, deleting, or changing a predicate; adding or deleting a local data variable; and changing a sequential segment. Structure changes include adding, deleting, or modifying a branch or a loop structure; and adding or deleting a sequential segment. The interface of a member function consists of its signature, access scope, and mode, its interactions with other member functions (for example, a function call). Any change in these elements is called an interface change of a member function.
- **Class change.** Direct modifications of a class can be classified into three types: component, interface, and relation changes. Any change in a defined/redefined member function or a defined data attribute is known as a component change. A change is said to be an interface change if it adds or deletes a defined/redefined attribute, or changes its access mode or scope. A change is said to be a relation change if it adds or deletes an inheritance, aggregation, or association relationship between the class and another class.
- **Class library change.** These include changing the defined members of a class, adding or deleting a class and its relationships with other classes, adding or deleting a relationship between two existing classes, and adding or deleting an independent class.

---

4 Changing a relationship \( R_1 \) (between two classes) into a relationship \( R_2 \) is considered as deleting \( R_1 \) and adding \( R_2 \).
Table 1. Different Type of Code Changes

<table>
<thead>
<tr>
<th>Components</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Change data definition/declaration/uses</td>
</tr>
<tr>
<td></td>
<td>Change data access scope/mode</td>
</tr>
<tr>
<td></td>
<td>Add/delete data</td>
</tr>
<tr>
<td>Method</td>
<td>Add/delete external data usage</td>
</tr>
<tr>
<td></td>
<td>Add/delete external data updates</td>
</tr>
<tr>
<td></td>
<td>Add/delete/change a method call / a message</td>
</tr>
<tr>
<td></td>
<td>Change its signature</td>
</tr>
<tr>
<td>Structure Changes</td>
<td>Add/delete a sequential segment</td>
</tr>
<tr>
<td></td>
<td>Add/delete/change a branch/loop</td>
</tr>
<tr>
<td>Component Changes</td>
<td>Change a control sequence</td>
</tr>
<tr>
<td></td>
<td>Add/delete/change local data</td>
</tr>
<tr>
<td></td>
<td>Change a sequential segment*</td>
</tr>
<tr>
<td>Class</td>
<td>Change a defined/ redefined method</td>
</tr>
<tr>
<td></td>
<td>Add/delete a defined/ redefined method</td>
</tr>
<tr>
<td></td>
<td>Add/delete/change a defined data attribute</td>
</tr>
<tr>
<td></td>
<td>Add/delete a virtual abstract method</td>
</tr>
<tr>
<td></td>
<td>Change as attribute access mode / scope</td>
</tr>
<tr>
<td>Relationship Changes</td>
<td>Add/delete a superclass</td>
</tr>
<tr>
<td></td>
<td>Add/delete a subclass</td>
</tr>
<tr>
<td></td>
<td>Add/delete as object pointer</td>
</tr>
<tr>
<td></td>
<td>Add/delete an aggregated object</td>
</tr>
<tr>
<td></td>
<td>Add/delete an object message</td>
</tr>
<tr>
<td>Class</td>
<td>Change a class (defined attributes)</td>
</tr>
<tr>
<td>Library</td>
<td>Add/delete a relation between classes</td>
</tr>
<tr>
<td>Class</td>
<td>Add/delete a class and its relations</td>
</tr>
<tr>
<td>Library</td>
<td>Add/delete an independent class</td>
</tr>
</tbody>
</table>

4.2 Data Change Identification

Data change identification is easy since the needed information is captured by the BBDs (and the internal representation) for the member functions.\(^5\) In particular, the information about each data item includes its access scope, type, access mode, update set (i.e., functions that define the data item), and use set (i.e., functions that use the data item). To identify data change, this information is compared with the information for the original software. If any of the above information is different, the corresponding type of change is identified.

4.3 Method Change Identification

We use method and member function interchangeably. Let \(BBD = (D_u, D_d, P, F, B)\) and \(BBD' = (D'_u, D'_d, P', F', B')\) be the BBDs for a member function \(C::f(\ldots)\) and its modified version \(C::f'(\ldots)\) respectively. Recall that \(B = (V, E)\) [or \(B' = (V', E')\)] is a directed graph that represents the control structure of \(C::f(\ldots)\) [or \(C::f'(\ldots)\)]. Method structure and/or component changes are identified as follows.

1. If \((V - V') \neq \emptyset\), then any \(v \in (V - V')\) is a deleted block node.
2. If \((V' - V) \neq \emptyset\), then any \(v' \in (V' - V)\) is an added block node.
3. If \((E - E') \neq \emptyset\), then any \(e \in (E - E')\) is a deleted control edge.
4. If \((E' - E) \neq \emptyset\), then any \(e' \in (E' - E)\) is an added control edge.

A member function interface change is identified as follows:

1. If \((D_u - D'_u) \neq \emptyset\), then some data uses are removed.
2. If \((D'_u - D_u) \neq \emptyset\), then some data uses are added.
3. If \((D_d - D'_d) \neq \emptyset\), then some data definitions are removed.
4. If \((D'_d - D_d) \neq \emptyset\), then some data definitions are added.
5. If \((F - F') \neq \emptyset\), then some function calls are removed.\(^6\)
6. If \((F' - F) \neq \emptyset\), then some function calls are added.

4.4 Class Change Identification

Changes in a class in an OO program can be classified into four types:

- **Class member changes.** A change in a data member includes the modification of its definition, declare-
tion, access scope, and access mode. A change in a member function includes statement changes, structural changes, such as changes on its control flow and data flow structures, and signature changes. The identification of these members has been described before.

- **Class interface changes.** A change is said to be an interface change of a class if it adds or deletes a defined member or modifies its access mode or scope. The identification of these changes is given below.

- **Class relation changes.** A change is said to be a relation change if it adds or deletes an inheritance, aggregation, or association relationship between the class and other classes. The identification of these changes can be performed by checking the structural differences between two ORDs. One is the original ORD for the classes in an OO program, and the other is the ORD for the classes in its modified version. The detailed description is given in the next section.

- **Object behavior changes.** Object behavior changes of a class C refers to changes in its OSD. It includes changes of atomic OSDs, such as changes on object states, transitions and paths, and changes in the object state hierarchy (or object state tree), such as adding or deleting a component of the OSD. The identification of object behavior changes in a class can be performed by comparing two different OSDs. One is the original OSD for a class, and the other corresponds to the modified class. The detailed description can be found in Gao et al. (1994).

A class can be view as a pair $C = (D_{def}, F_{def})$, where $D_{def}$ is a set of defined/redefined data attributes and $F_{def}$ is a set of defined/redefined member functions. Let $C' = (D'_{def}, F'_{def})$ be a modified version of a class $C$. Then class code change is identified as follows:

- If $(D_{def} - D'_{def}) \neq \emptyset$, then any $d \in (D_{def} - D'_{def})$ is a deleted data attribute.
- If $(D'_{def} - D_{def}) \neq \emptyset$, then any $d \in (D'_{def} - D_{def})$ is an added data attribute.
- If any $d \in D'_{def} \cap D_{def}$ is changed, then a residual data attribute is changed.
- If $(F_{def} - F'_{def}) \neq \emptyset$, then any $f \in (F_{def} - F'_{def})$ is a deleted member function.
- If $(F'_{def} - F_{def}) \neq \emptyset$, then any $f \in (F'_{def} - F_{def})$ is an added member function.
- If any $f \in F'_{def} \cap F_{def}$ is changed, then a residual defined/redefined member function is changed.

### 4.5 Class Library Change Identification

A class library $L$ is a collection of ORDs. An ORD is an edge-labeled directed graph $ORD = (V, L, E)$, where $V$ is the set of nodes representing the object classes, $L = \{I, Ag, As\}$ is the set of edge labels (for inheritance, aggregation, and association), and $E = E_I \cup E_{Ag} \cup E_{As}$ is the set of edges. Modifications to a library can be classified into three basic cases, namely, adding an ORD, deleting and ORD, and changing and ORD. In the first two cases, there is no impact on the other ORDs; therefore, we consider only the last case. An ORD can be changed in several ways: changing the defined members of a class, adding or deleting a relation between two existing classes, and adding or deleting a class and its relations. Change identification for a single class has been discussed in the previous subsection. Here, we focus on structure change of a class library.

Let $ORD = (V, L, E)$ and $ORD' = (V', L', E')$ be the ORDs for two versions of the same software. A structure change in an ORD is identified as follows:

- If $(V' - V) \neq \emptyset$, then any $v \in (V' - V)$ is an added class node.
- If $(V - V') \neq \emptyset$, then any $v \in (V - V')$ is a deleted node.
- If $(E' - E) \neq \emptyset$, then any $e \in (E' - E)$ is an added new edge.
- If $(E - E') \neq \emptyset$, then any $e \in (E - E')$ is a deleted edge.
- If any $v$ in $V \cap V'$ is changed, then a residual class is changed.

### 5. CLASS FIREWALL

After an change is made in an OO program, regression testing involves retesting components at different levels: class members, class/objects, subsystems, and systems. To save effort and time, regression testing needs to retest only those components affected by the modification. There are various levels of changes, such as data member change, function member change, class change, and relation change. Each of these changes has different effects on the modified program. Here we define class firewall as a...
mechanism to identify the effect of a class change at the class level.

5.1 The Notion of a Class Firewall

Any class can be changed in many ways. Here we can classify various changes into three types:

1. They affect its object behaviors, such as states or transitions.
2. They affect the operations and behavior of its member functions.
3. They affect the relationships between this class and others.

Intuitively, a class firewall for a class C, denoted as $CFW(C)$, in an OO program or library is the set of classes that could be affected by changes to class C. These affected classes should be retested when class C is changed. Here, we consider only the first two types of changes. In other words, we assume that the relations between classes are not affected by the changed class. The problem associated with class relation changes will be addressed elsewhere (Gao 1995). We wish to point out that the notion of class firewall defines the classes that are possibly (not necessarily) affected by the changes in class C.

Lemma 1. Let class A be a subclass of class B in the inheritance hierarchy, and only class B is changed without affecting its relationships with other classes. If the change affects the inherited members (from class B) of class A, for adequate testing, not only should class B be unit retested, but class A should also be retested and reintegrated with class B.

Proof. Since class A is a subclass of class B, class A must inherit some of class B’s attributes. Thus, class A has a code dependency on class B due to the inherited attributes. Because the modification made in class B affects the inherited members of class A, according to Perry and Kaiser (1990), they should be retested in class A to ensure that they work well in its reused context. Next, they should be reintegrated with other attributes of class A to make sure that they work well together. In addition, classes A and B should be reintegrated to make sure that correct member functions are executed in the inheritance hierarchy. This is necessary when dynamic binding and polymorphism occurs.

Lemma 2. Let class A be an aggregate class of class B, and class B is changed without affecting its relationships with other classes. For adequate testing, not only should class B be unit retested, but class A should also be retested with class B.

Proof. Because a change of class B affects its object behavior or the operations of its member functions, class B should be retested at the unit level. Since class A contains some objects of class B as its component, the change will affect class A’s object behavior in the following aspects: the behavior of the aggregated part (e.g., class B’s objects) is affected, and other members of class A are affected if they are directly (or indirectly) dependent on the aggregated part. Thus, those dependent members of class A should be retested and reintegrated with the instances of class B. Moreover, class A’s object behavior should be retested.

Lemma 3. Let A and B be two independent classes, and class A is associated with class B. If class D is changed without affecting its relationships with other classes, then for adequate testing, not only should class B be unit retested, but class A should also be retested and reintegrated with class B.

Proof. Since class A is associated with class B, one (or more) of the following cases must occur: 1) at least one member function of class A depends on some data members of class B; 2) at least one member function of class A has a message passing to class B; and 3) at least one member function of class A passes some of class B’s objects as parameters in message passing. In all three cases, one (or more) member(s) of class A has (have) some data dependency on some of class B’s members or objects. If the change in class B directly (or indirectly) affects those members or objects on which class A’s members depend, then after class B retesting, these dependent members of class A should be retested with class B. Otherwise, these members should be reintegrated with other members of class A. Therefore, after changing class B, class A should be retested and reintegrated with class B to ensure adequate testing.

5.2 Constructing a Class Firewall

To compute the class firewall, we first introduce a binary relation $R$ that is derived from the directed edges of an $ORD = (V, L, E)$:

$$R = \{(C_2, C_1) | C_1, C_2 \in V \land (3l)$$

$$l \in L \land (C_1, C_2, l) \in E)\}$$

* A class is called an aggregated class if its object contains other class objects as its components.

* The dependency may be a data, state, message, or control dependency.

* The message passing can be static or dynamic.
We call $R$ the dependence relation since it defines the dependence between the classes, according to the inheritance, aggregation, and association relations. More specifically, $(C_2, C_1) \in R$ if and only if one of the following cases holds: 1) $C_1$ is a derived class of $C_2$; 2) $C_1$ is an aggregate class of $C_2$, that is, $C_2$ is part of $C_1$; or 3) $C_1$ is associated with $C_2$ either by accessing its data members or passing some messages. In all these cases, $C_1$ is dependent on $C_2$ in the sense that code changes to $C_2$ affect the behavior of $C_1$. The computed class firewall for a class $C$, denoted $CCFW(C)$, then is defined as

$$CCFW(C) = \{C| (C, C_i) \in R^*\}$$

where $R^*$ is the transitive closure of $R$. That is, if $(C_i, C_j) \in R$ and $(C_j, C_k) \in R^*$, then $(C_i, C_k) \in R^*$. The transitive closure of $R$ can be computed by the algorithm (Aho et al., 1983).

**Theorem 1.** Let $G$ be an ORD of a given OO program $P$, and let $R$ be the dependence relation derived from $G$. Let $C$ be a class in which a change is made to its define or redefined members. Assume the dependencies between the classes of $P$ are the dependencies of inheritance, aggregation, and association relations. Then $CCFW(C) = CFW(C)$, that is,

1. $CCWF(C) \subseteq CFW(C)$.
2. $CFW(C) \subseteq CCFW(C)$.

**Proof 1:** We need to prove that for any class $C_i \in CCFW(C)$, class $C_i$ could be affected by the changes to class $C$ and should be retested. The proof is by induction on $n$ of dependence relation $R^n$.

**Basis:** For any class $C_j$, $(C_j, C_i) \in R^1$. Then $(C_j, C_i) \in E$ of $G$ based on the definition of $R$. Thus, $(C_j, C_i, l)$ must be one of the three cases: inheritance edge, aggregation edge, or association edge. According to Lemmas 1-3, in all of these cases, $C_j$ could be affected by changes to $C$ and $C_i$ should be retested.

**Inductive hypothesis:** Assume that for any class $C_j$, $(C_j, C_i) \in R^k$. Then class $C_j$ would be affected by changes to class $C$ and should be retested.

**Inductive step:** We need to prove that for any class $C_i$, if $(C, C_i) \in R^{k+1}$, then class $C_i$ could be affected and should be retested.

According to the transitive nature of $R$, there must be $(C_j, C_i) \in R$ and $(C_j, C_k) \in R^k$. From the inductive hypothesis, class $C_i$ could be affected by the changes to class $C$ and should be retested. From the basis, class $C_i$ could be affected by the changes of class $C_j$ and should also be retested. Thus, class $C_i$ could be affected by the changes to class $C$ and should be retested.

**Proof 2:** We need to prove that if any class $C_i$, $C_i \in CFW(C)$, then class $C_i \in CCFW(C)$. We prove this by contradiction. Assume that there is class $C_j$ which is not in $CCFW(C)$, but $C_j \in CFW(C)$ in the sense that it could be affected by the changes to class $C$ and should be retested. Class $C_j$ can be affected by the change of class $C$ only when $C_j$ is either directly or indirectly dependent on the defined members of class $C$. Thus, there must be a dependency chain (denoted \textit{CHAIN}): $(C_1, C_2, \ldots, C_k, C)$, from class $C_1$ to class $C$. As we know, the types of dependencies between two classes/objects can be classified into code dependency, object dependency, control dependency, and data or state dependency. According to the assumption, they are dependencies of the three relations: inheritance, aggregation, and association. Hence, $CHAIN \subseteq R$ and $(C_i, C) \in R^k$. Therefore, $(C_i, C) \in CCFW(C)$. This a contradiction. QED.

From Theorem 1, the following theorem can be easily derived.

**Theorem 2.** Let $CCFW(C)$ be the computed firewall for class $C$. For any class $C_i$, if $C_i$ is not in $CCFW(C)$, the $C_i$ is not affected by the change in class $C$, and hence, no retest is needed.

Using Theorem 1, we can construct the class firewall for a changed class to enclose all possible affected classes that should be retested. According to Theorem 1, these classes should also be reintegrated with their subclasses, aggregated classes, and associated classes to achieve adequate testing. The detailed results on class reintegration test strategy are provided in the next section. The notion of class firewalls can be extended to a set of changed classes. Let $S = \{C_1, C_2, \ldots, C_k\}$ be a set of classes that are changed. Then, the class firewall for $S$, also denoted $CFW(S)$, is defined as follows:

$$CFW(S) = \bigcup_{C \in S} CFW(C)$$

As an example, Figure 3 depicts an ORD for a subset of classes in the InterViews library, and Figure 4 depicts the class firewall for class Subject in the subset.

6. **TEST ORDER FOR CLASS FIREWALLS**

After identifying the class firewall for a changed class, the tester needs a cost-effective strategy to conduct the retests for each class at the class-unit level and reintegrate classes at the class integration
level. The existing test strategies, including top down, bottom up, and sandwich, rely on the tester to make the selection. They conduct the unit test and integration test by using test stubs and test drivers.

However, in regression testing of OO programs, it is costly and difficult to construct test stubs and drivers for a class object or a class function member, for the following reasons. First, simulating a class or its objects is very costly and difficult because the tester not only has to understand and simulate its class members, object states, and behaviors, but also has to understand and simulate its inherited members and component class objects. Second, simulating a member function is very expensive and complicated because the tester has to understand chains of member function invocations and simulate the different state changes of class objects and messages between the class objects due to these function invocations\(^{12}\) to construct appropriate test stubs. In addition, according to Wilde and Huitt (1991), 80% of member functions consist of one or two statements. Thus if we can test components using tested components by following a proper test sequence, then the test effort for constructing test stubs and test drivers can be reduced. The main idea is to test the independent components first and then test the dependent components based on their relationships. For example, in testing the called functions before the calling function, the effort required to construct the test stub for the calling function can be saved. Similarly, in testing a base (or aggregated) class before the subclasses (or assembly classes), the stub for the base classes (or aggregated classes) can be reduced for testing the subclasses (or assembly classes). Moreover, the test information for the base class (or aggregated class) can be reused in testing its subclasses (assembly classes).

To resolve this problem, we introduce a test strategy, called test order, which serves as a detailed road map in class unit retesting and class reintegration testing.

The test order problem for class firewalls can be stated as finding a desirable order for testing the classes that are affected by code changes to a set of classes. By testing of a class, we mean structure testing, function testing, object-state testing, and/or data flow testing of the member functions of the class. Exactly how these tests are performed is beyond the scope of this article.

A desirable test order also implies effective reuse of previously generated test cases in the new, reusing context. In Harrold et al. (1992), a incremental methodology is proposed for reusing test informa-

---

\(^{12}\) Unlike a procedure in conventional programs, a function member of a class cannot be viewed as a function from its input domain to its output domain. Instead it only changes the states of class objects or passes the message between objects.
tion in retesting inherited members of classes in a class hierarchy. Our result on test order supplements their work and allows test cases to be reused in a more general context, where inheritance, aggregation, and association relations exist.

The above discussion implies that a desirable test order is one that requires minimum effort to construct the test stubs. Since the effort to construct the test stubs differs substantially from case to case, we assume that the total effort is proportional to the number of stubs that need to be constructed. The reader will see later than this assumption does not affect the usefulness of the method, since it can be easily tailored to take into consideration the effort required to construct individual stubs.

A solution to the test order problem must consider two cases:

1. The \( \text{ORD} = (V, L, E) \) is an acyclic digraph, meaning that there exists no cycle in the digraph.
2. The \( \text{ORD} = (V, I, E) \) is a cyclic digraph, meaning that there exists one or more cycles.

In the first case, the test order is simply the topological sorting of \( CCFW(S) \) using the dependence relation \( R \) (defined in Section 4) as the precedence relation. The computational complexity is \( O(|CCFW(S)|) \), that is, the number of classes in the firewall for \( S \), since each node needs to be visited only once. The effort required to construct the test stubs is zero and hence is minimal. The solution to case 2 is nontrivial since topological sorting cannot be applied to cyclic digraphs. The remainder of this article is devoted to developing a solution for this case.

6.1 Overview of the Test Order Finding Algorithm

The algorithm is based on two key concepts. The first is the notion of a cluster, which is a maximal set of vertices that are mutually reachable though the relation \( R \) defined in Section 4.\(^{13}\) A cyclic \( \text{ORD} \) may have more than one cluster, and a cluster may contain only one vertex (such clusters are called unit clusters).

---

\(^{13}\) This is also called a strongly connected subgraph in graph theory.
clusters). A cyclic digraph $ORD = (V, L, E)$ can be transformed into an acyclic digraph $ORD' = (V', L', E')$ in which $V'$ is the set of clusters in the $ORD$, $E'$ is the set of edges between clusters in the $ORD$, and $L'$ is the set of labels on edges in $E'$. It can be proved that $ORD'$ must be acyclic; otherwise, some of the clusters must not be a maximal set of mutually reachable vertices. We do not pursue a proof here due to space limitations. Since $ORD'$ is acyclic, topological sorting can be applied to produce a test order for the clusters. The test order is called the major test order.

The second notion is cycle breaking, that is, to identify and temporally remove an edge(s) from a nonunit cluster so that the vertices of the cluster and their associated edges form an acyclic subgraph. Again, topological sorting can be applied to the acyclic subgraph to derive a test order, called the minor test order. Thus, a cyclic $ORD$ can be tested first according to the major test order and then the minor test order.

We are now ready to outline the algorithm:

**Step 1.** Transform the $ORD$ into an acyclic digraph $ORD'$ by grouping class nodes into clusters. Consider each cluster as a node in the $ORD'$. 

**Step 2.** Produce a test order (called major test order) of the $ORD'$ using the topological sorting algorithm (Aho et al., 1983).

**Step 3.** For each nonunit cluster of $ORD'$ do steps 4 and 5.

**Step 4.** For each cycle of the cluster, select and remove an edge to break the cycle.

**Step 5.** Produce a topological sorting for each resulting cluster (acyclic subdigraph) in step 4, and generate a test order (called the minor test order) for the classes in each cluster.

The above algorithm applies to acyclic as well as cyclic digraphs. If the digraph is acyclic, then steps 3–5 are not performed. Only steps 1 and 4 require elaboration since the other steps are straightforward. Therefore, in the following sections, we focus on converting a cyclic digraph to an acyclic one and strategies for selecting an edge to break a cycle.

### 6.2 Converting a Cyclic $ORD$ to an Acyclic $ORD$ Using Clusters

The algorithm uses the transitive closure of the dependence relation $R$ defined in Section 3. As usual, the transitive closure is denoted $R^*$. For any two vertices $u, v \in V$, where $V$ is the vertex set of the original cyclic $ORD = (V, L, E), \langle u, v \rangle \in R$ means code change to $u$ would affect the behavior of $v$, and $\langle u, v \rangle \in R^*$ means $v$ is transitively affected by code change to $u$. Clearly, if $\langle u, v \rangle \in R^*$ and $\langle v, u \rangle \in R^*$, then changes to $u$ affect $v$ and vice versa. In this case, $u$ and $v$ are mutually reachable and they are in a cycle. Therefore, $u$ and $v$ must be placed in the same cluster. Since the mutually reachable relation is an equivalence relation (i.e., reflexive, symmetric, and transitive), we follow the usual mathematical convention and denote the cluster that contains $u$ and $[u]$. The purpose of this algorithm is to identify such pairs, place them in a cluster, and compute the edges that define the precedence relation between the clusters.

**Input:** $ORD-(V, L, E)$

**Output:** an acyclic digraph $ORD' = (V', L', E')$, where $V' \subseteq \{v'|v' \in 2^V\}$ $L' \subseteq \{[v']|[v'] \in 2^V\}$ $E' = \{(u', u', l')|([u'] \times [v'] \times 1') \cap E \neq \emptyset\}$

**Step 1. Initialize.** Let $[v] \in V'$ for every $v \in V$, $L' = E' = \emptyset$, and each vertex of $V$ is marked unclustered (the marking will be changed to clustered when the vertex is placed into an established cluster). We assume that the vertices of $ORD$ are indexed as $v_1, v_2, \ldots, v_n$. Any indexing scheme can be used, since it is not essential.

**Step 2. Compute $R^*$.** Compute the transitive closure, denoted $R^*$, for $R = \{(v, v')|v, v' \in V \land (\exists l)(l \in L \land \langle v, v', l \rangle \in E)\}$

**Step 3. Compute clusters.** For $i = 1$ to $n - 1$, do the following if $v_i \in V$ is marked unclustered. For $j = i + 1$ to $n$, if $\langle v_i, v_j \rangle \in R^*$ and $\langle v_j, v_i \rangle \in R^*$, then

1. insert $v_j$ into the cluster containing $v_i$, that is, set $[v_i] = [v_i] \cup \{v_j\}$
2. mark $v_j$ as clustered
3. delete $[v_j]$ from $V'$, i.e., set $V' = V' - \{[v_j]\}$

**Step 4. Compute the edges $E'$.** For each pair $u', v' \in V'$, if there is a directed edge from some vertex in $u'$ to some vertex in $v'$ in the original digraph, then create a directed edge from $u'$ to $v'$, with a label of the union of the labels of the original edges. This is formally computed by the following formulas:

$E' = \{\langle u', v', l' \rangle|u', v' \in V' \land l' \subseteq L' \land (u' \times v' \times l') \cap E \neq \emptyset\}$

Figure 4 is a cyclic $ORD$ for a subset of InterViews library, and Figure 5 shows the acyclic $ORD'$ produced by this algorithm.

---

14 We give algorithms for step 1 in the next section.
6.3 Breaking a Cycle

Given a cyclic ORD, such as the vertices and the associated edges in a cluster, how can one determine a test order to test the classes that are mutually dependent on each other? Our answer to this question is to break the cycle(s) by temporally removing some of the directed edges. This leads to the second question: which edge(s) are to be removed? Our answer is to remove association edges since, among the three relations discussed here, association relation represents the weakest coupling between the two related classes. The other two relations, namely, inheritance and aggregation, involve not only control coupling, but also code dependence and data coupling. The following theorem provides the basis for cycle breaking of association edges:

**Theorem 3.** Every directed cycle of an ORD contains an association edge.

The proof of the theorem is lengthy and can be found in Gao (1993). The basic idea is that inheritance and aggregation represents a part-of relation between the data spaces of the objects of the two related classes (top-down part-of for inheritance, and bottom-up part-of for aggregation). Moreover, these two relations are transitive relations. Object creation at run time leads to data space explosion and causes undesired side-effects.

If there is more than one association edge that can be removed to break a cycle, then the one that is removed affects the test effort. One possible solution is to use the cardinality concept for ORDs to represent the number of class members that are involved in the association relation between two classes. However, we do not pursue this issue here. Our current implementation simply picks up any association edge and removing it until no cycle exists.

After breaking the cycles in each cluster, topological sorting can be applied to produce a test order. Figure 6 shows the major test order and minor test order for the InterViews examples shown in Figure 5. The test order suggests that the classes be tested in the following order: `Subject`, `InteractorTr`, `Canvas-Rep`, `ButtonState`, `Canvas`, `Sensor`, `Scene`, `World`, `Event`, `ButtonList`, `Button`, `MonoScene`, `TextButton`, `ControlState`, `Control`.

Since the generated test order is a topological order, it is possible that two or more vertices may have the same number. For example, the vertices `Canvas` and `ButtonState` in Figure 6 have the same major test order, which is 2. In this case, either of them can be tested before the other.

6.4 Test Order-Based Regression Test Strategy

**Class unit retest strategy.** For any change in a class library, a class firewall for class C is constructed...
based on the given ORD, the dependent relation $R$ (Between class $C$ and other classes), and the transitive closure $R^*$. Thus, the relationship(s) between these classes (including class $C$) in a class firewall is actually a subgraph of the ORD.

During regression testing, the previous test order algorithm can be used to find a cost-effective test order for the classes in a class firewall. Then, the generated test order is used as a guideline to select individual classes to conduct retests at the class unit level. Figure 7 shows the generated test order of the firewall for class Subject in Figure 4. The testing is carried out using the major test order to test each cluster of classes. If the cluster to be tested is a unit cluster, then the member functions of the class in the cluster are tested according to their invocation dependencies. That is, if member function $g$ invokes member function $f$, then $f$ is tested before $g$. If recursive calls do not exist among the member functions, then topological sorting can be applied to produce a test order for testing the member functions. Otherwise, the termination condition for the recursive calls is identified and tested first, followed by testing of the other cases recursively. We do not address member function test order, interprocedural testing, and testing of recursive functions in this article. The interested reader is referred elsewhere (Hwang et al., 1988; Harrold and Malloy, 1991).

If the cluster to be tested is a nonunit cluster, then the classes in the cluster are tested according to the minor test order. Since the classes are cyclically dependent on each other, it is not possible to test all the member functions of each of the classes according to the minor test order in one pass. Therefore, the classes are tested in two or more passes. The trick is to test, in each pass, the member functions of a class that do not invoke other untested member functions. This process is repeated until all the member functions are tested. If recursion does not exist among the member functions of the classes, then two passes suffice; otherwise, recursive, multiple passes are needed.

The retest for a class may include function member testing (both white box or black box), object state testing, and data flow testing. The purpose is to check the different aspects of the target class, including its function members, objects states and behaviors, and data flows.

**Test strategy for class reintegration.** After retesting classes (in a class firewall) at the class unit level, we need to reintegrate these classes. The generated test order for a class firewall can be used as a guideline.

---

15 A nonunit cluster consists of more than one class.
for class reintegration. In this section, we provide an incremental class reintegration strategy and detailed algorithm for conducting a cost-effective class reintegration. As in our class integration strategy, the ORD model and test order information are used to find an ideal test sequence of class reintegration, so that the maximal tested components are used and the number of necessary test stubs is reduced.

In class reintegration, the major tasks are as follows:

- Retest the interclass relationships between classes in the firewall. For example, retest an inherited member in its reuse contexts if it is changed in its defined class or its reuse contexts are changed.
- Retest the interclass relationships between classes inside and outside the firewall.
- Reintegrate state-dependent object behaviors between different class objects if they are changed or affected.

Let $G = (V, E, L)$ be an ORD, $C$ be a changed class, and $G' = (V', E', L')$ be the ORD corresponding to the class firewall $CCFW(C)$.

The algorithm for incremental class reintegration:

Assume: (a) $S$ be the class integration suite.
(b) $T(v_k) \in [1, N]$ is the major test order number of class $v_k$ in the firewall.

(c) $t(v_k)$ is a minor test order number for class $v_k$ in the firewall. If $t(v_k) = 0$, then $v_k$ is a unit cluster.

step 1: Build the initial integration suite $S$: $S = V - V'$
/* put all of the unaffected and resulted classes of $G$ into $S$*/

step 2: For $i = 1$ to $N$ do:
$\forall v_x, v_y \in V' (T(v_x) = i)$ do:
if $t(v_x) = 0$
then: /* $v_x$ is a unit cluster.*/
(a) For every edge $e = (v_x, v_y, l) \in E$ and $v_x \in S$ do: reintegrate $v_x$ and $v_y$.
(b) put $v_x$ into suite $S$.
else: $v_x$ is a class node in a cluster.
(a) find other classes inside of the cluster, and reintegrate them with $v_x$.
(b) For any class $v_u$ inside the cluster do:
(ii) put $v_u$ into suite $S$.

Apply the reintegration algorithm on the class firewall in Figure 4. In the first step, the initial integration suite $S$ is constructed using those unchanged
and unaffected classes. In the second step, the classes (Subject and InteractorItr) in cluster 1 are reintegrated and added into S in the first loop (when i = 1). During the second loop, class ButtonState is reintegrated with class Subject and added into S. In addition, the classes (Control and ControlState) in cluster 2 are reintegrated, and then they are reintegrated with the classes in S, such as MonoScene, and finally they are added into S. Class TextButton and cluster 3 are reintegrated with S and added into S in the third and fourth loops, respectively.

The purpose of class reintegration testing is to check if the classes, which are related to each other or dependent on each other, can work together properly. This testing focuses on inherited members, aggregated parts, state-dependent behaviors, and message passing between different objects.

7. AN EXAMPLE

We have implemented an OO testing environment using the reverse engineering approach (Kung et al., 1993). In the testing environment, a test support tool, called the firewall identifier, is developed based on the class firewall concept to automatically identify possibly affected classes for a changed class. After a class firewall is recognized, a test order generator, which is designed based on the results of this work, is used to find the desired cost-effective test order for retesting the affected classes. The generated test order provides not only a cost-effective class unit retest strategy, but also a detailed class reintegration strategy.

We have applied this tool to many applications, including the InterViews library. The InterViews library contains 147 files, > 140 classes, and > 400 relationships between these classes (see Table 3). Without automatic support, it is very difficult for a regression tester to identify the relations between the classes.

How should regression testing for this library be conducted when changes are made to one or more classes? For example, when a change is made in class Canvas, the regression tester faces the first challenge of identifying the affected classes. The tester has only two choices: either retest all classes or manually find out those affected classes statically (or dynamically) during the tests. Both approaches are costly and undesirable.

The firewall identifier program of the OO testing environment provides an automatic tool for finding possible affected classes for retesting. For a changed class Canvas, the identified class firewall is shown in Table 2.

After identifying the class firewall for a changed class, the tester starts to conduct the retests for the classes in the firewall. The first task is to retest each affected class at the class unit level. But which class should be first? Without a detailed guideline and automatic support tools, the tester will encounter another challenge: very costly test stubs. For example, if the Button class is chosen to be tested first,
Table 3. Relationships in the Interview Library (Version 2.6)

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Classes</td>
<td>144</td>
</tr>
<tr>
<td>Inheritance Relationships</td>
<td>104</td>
</tr>
<tr>
<td>Aggregation Relationships</td>
<td>29</td>
</tr>
<tr>
<td>Association Relationships</td>
<td>290</td>
</tr>
</tbody>
</table>

then, as shown in Figure 4, test stubs for classes `Sensor`, `ButtonState`, `Event`, and `ButtonList` are needed. In constructing the test stub for `ButtonState`, the tester needs to understand other classes, such as `Subject` and `InteractorItr`, because of the relationships between them. This suggests that a cost-effective strategy is needed to retest classes at the class unit level. A similar problem is encountered by the tester when he or she performs the class reintegration tests. Thus, the real challenges for the tester are 1) what is a cost-effective strategy for class unit retests? 2) What is a desirable strategy for class reintegration testing?

The test order generator in the OO testing environment automatically generates a cost-effective test order for the classes in the firewall, so that the cost in constructing test stubs for class unit regression testing and class reintegration testing is reduced. Table 4 lists a generated cost-effective test order for

Table 4. Test Order of the Firewall of Class Subject

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Major Order</th>
<th>Minor Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canvas</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RasterRep</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Raster</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Painter</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Event</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Sensor</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Interactor</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>TopLevel</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Scene</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>World</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>ViewList</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Perspective</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Adjuster</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Banner</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Border</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>BoxElement</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Card</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Glue</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MonoScene</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Rubberband</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Scroller</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Slider</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TestDisplay</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Tray</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>WorldView</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Subject</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>InteractorItr</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Box</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Deck</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Frame</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>GrowingVertices</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>HBorder</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>HGlue</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>HScrollbar</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Mover</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Panner</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>RotatingRect</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>RubberEllipse</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>RubberGroup</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>RubberLine</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>RubberPointList</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>RubberRect</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>TextEditor</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>VBorder</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>VGlue</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>VScrollbar</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Viewpoint</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Zoomer</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ControlState</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Control</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Dialog</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
all classes in the firewall for class Canvas. During the class unit retest, the generated test order can be used as a strategy to help the tester find a cost-effective test sequence. Later, the test order can be used as a road map for class reintegration testing to reduce the test cost.

Now the only question is how effective the computed test orders are. In Figure 8, we provide our experimental results on the cost of class/object tests stubs using the InterView library. In the experiment, we compute the number of class/object stubs needed during class unit testing for all classes in two different ways. In the first approach, we use a random test order in the class unit testing to find the number of stubs needed for dependent classes/objects. By applying 100 random test orders (or test sequences), we found that the average number of stubs needed in 192.23 for each test order. However, in the second approach, a computed test order is used as a road map for class unit testing. The result shows that the total number of stubs needed for the InterView library is eight when we follow the computed test order to conduct the class unit test.

8. CONCLUSIONS AND FUTURE WORK

We have discussed the regression processes and addressed problems in regression testing of OO programs. We have presented methods for finding the affected classes when changes are made to classes of a C++ program. We have also presented a test order generation algorithm that provides a cost-effective test sequence of components, which serves as a road map for a tester to retest classes in the class firewall. The results can be used to conduct cost-effective regression testing of classes at the class unit and class integration levels. These techniques have been incorporated into our OO testing environment and have been applied to realistic applications, like the InterViews class library. The results so far are promising. We are currently investigating guidelines for finding a firewall of a changed class member and algorithms for testing the member functions of the classes.

---

16 We assume that each stub for a class (or object) simulates the behavior of the class object.

![Figure 8. The experiment results on the cost of class/object test stubs.](image)
ACKNOWLEDGMENT

The material presented here is based on work supported by Texas Advanced Technology Program grant 003056-097, Fujitsu Network Transmission Systems, Inc., Hewlett-Packard Company, and the Software Engineering Center for Telecommunications at the University of Texas at Arlington.

REFERENCES


