The Implementation of an Extensible System for Comparison and Visualization of Class Ordering Methodologies

Nicholas A. Kraft [a] Errol L. Lloyd [b] Brian A. Malloy [a]
Peter J. Clarke [c]

[a] *Computer Science Department, Clemson University, Clemson, SC 29634*

[b] *Department of Computer and Information Sciences, Delaware University, Newark, DE 19716*

[c] *School of Computer Science, Florida International University, Miami, FL 33199*

Abstract

In this paper we present the design and implementation of a system that exploits well-known design patterns to facilitate construction of an extensible system for comparison and visualization of ordering methodologies for class-based testing of C++ applications. Using our implementation, we present a comparative study and evaluation of two advanced ordering methodologies: the edge based approach by Briand, et al., and the Class Ordering System (COS) introduced in this paper. We compare two variations of the approach by Briand and three variations of the COS system and draw conclusions about the number of edges removed, and therefore the number of stubs that must be constructed, using each approach. We also compare the run-time efficiency of each approach and raise some interesting questions about edge type considerations for removal in the presence of cycles in the ORD. Using the design patterns together with the *dot* tool from the Graphviz package, we incorporate visualization of the ORD and the edge removals into our system. We present details and graphical visualization of the edge removal process.

Key words: Object Relation Diagram (ORD), class-based testing, design patterns, command pattern, visitor pattern, singleton pattern, visualization, Unified Modeling Language (UML)

Email addresses: `nkraft@cs.clemson.edu` (Nicholas A. Kraft), `elloyd@mail.eecis.udel.edu` (Errol L. Lloyd), `malloy@cs.clemson.edu` (Brian A. Malloy), `clarkep@cs.fiu.edu` (Peter J. Clarke).
URLs: `www.cs.clemson.edu/ nkraft` (Nicholas A. Kraft), `www.eecis.udel.edu/ elloyd` (Errol L. Lloyd), `www.brianmalloy.com` (Brian A.

1 Introduction

The current trend in the construction of large scale systems is to exploit object technology to facilitate reuse, extensibility, maintenance and testing of such systems. However, testing a complete object-oriented system is a formidable task with some imposing associated problems. One problem is that the application must be available before system testing can begin and this may be late in the life cycle. A second problem is that during the testing of a large system there are risks of complex interactions among errors with a concomitant destabilization of the corrected classes or components. Finally, system level testing may be too coarse-grained to permit the tester to meet the adequacy criteria for some regions of the code (Binder, 2000; Lloyd and Malloy, 2005).

To address these problems many developers prefer a progressive approach where individual classes are tested early in the development process. Indeed, some methodologies advocate the use of mock objects to facilitate testing before design (Martin, 2003). Also, class-based testing attempts to isolate errors to avoid interactions among the errors and to allow the developer to test more thoroughly in an effort to meet the adequacy criteria established for each class or class cluster. In general, however, testing techniques are heuristics and their performance varies with different scenarios; thus, there is an identified need for empirical evaluation and comparison of testing strategies (Do et al., 2004; Harrold, 2000; NIST, 2002; Orso et al., 2004).

One difficulty in class-based testing is that classes interact with other classes; therefore, a fundamental issue in testing object-oriented systems is the determination of an integration order for the classes. To determine an order for class-based testing, previous approaches have constructed an Object Relation Diagram, ORD, whose nodes are classes and whose edges represent the relationships between the classes (Kung et al., 1995; Labiche et al., 2000; Malloy et al., 2003a; Tai and Daniels, 1997). Early research included three (Kung et al., 1995; Tai and Daniels, 1997) and then four edge types (Briand et al., 2002; Labiche et al., 2000) in the ORD. However, to accommodate the full complement of C++ language features, including polymorphism, templates, and nested classes, we extend the ORD to include six edge types. Since the ORD is a class diagram (Labiche et al., 2000), five of the edge types are taken from the UML specification (OMG Unified Modeling Language Specification, 2003); the sixth edge type is taken from (Labiche et al., 2000). If there are no cycles in the ORD, then a reverse topological ordering of the nodes will yield a test order that obviates the construction of stubs.

However, in the presence of cycles in the ORD, one or more edges must be removed and, to test a client class that uses an untested supplier class, stubs

Malloy), `www.cs.fiu.edu/clarkep` (Peter J. Clarke).

must be constructed to simulate the behavior of the untested supplier class. Several researchers have maintained that the most costly aspect of class-based testing is the construction of stubs, since it is not always feasible to construct a stub that is simpler than the code it simulates and stub generation cannot be fully automated (Lloyd and Malloy, 2005; Kung et al., 1995; Labiche et al., 2000). The problem of removing a minimum number of edges to eliminate cycles in an ORD is equivalent to the *feedback arc set problem*, which has been shown to be NP-complete (Garey and Johnson, 1979; Karp, 1979). Nevertheless, since stub construction can be the most difficult and expensive aspect of class-based testing, several ordering methodologies have been proposed for breaking cycles in an ORD as a component of class-based testing (Briand et al., 2001; Kung et al., 1995; Labiche et al., 2000; Tai and Daniels, 1997). Those methodologies differ in: what assumptions they make about how cycles may be broken; how they break *ties* when there are several equivalent options for breaking cycles; and precisely what is stubbed once a cycle is broken. In particular, all but one of the methodologies are based on removal of edges to break cycles (Briand et al., 2001; Kung et al., 1995; Labiche et al., 2000; Malloy et al., 2003a), and one methodology is based on removal of nodes (Tai and Daniels, 1997). Of the methodologies based on edge removal, the methodology of Briand et al. (Briand et al., 2001) subsumes that of Kung et al. (Kung et al., 1995), and Briand et al. (Briand et al., 2001) demonstrated that their methodology is better than (Labiche et al., 2000).

In this paper we present the design and implementation of a system that exploits several well-known design patterns to facilitate construction of an extensible system for comparison and visualization of ordering methodologies for class-based testing of C++ applications. Using our implementation, we present a comparative study and evaluation of two advanced ordering methodologies: the edge based approach by Briand, et al. (Briand et al., 2001), and the Class Ordering System (COS) approach that we introduce here. We compare two variations of the approach by Briand and three variations of the COS system and draw conclusions about the number of edges removed, and therefore the number of stubs that must be constructed, using each approach. We also compare the run-time efficiency of each approach and raise some interesting questions about edge type considerations for removal in the presence of cycles in the ORD. Our study addresses the identified need for empirical evaluation and comparison of testing strategies. Using the design patterns together with the *dot* tool from the Graphviz package (AT&T Labs, 2005), we are able to incorporate visualization of the ORD and the edge removals into our system; thus, we also present details and graphical visualization of the edge removal process, which greatly facilitates comprehension, debugging and validation of the generated ORD and the cycle breaking process.

The contributions of this paper are as follows:

(1) We provide the design and implementation of an extensible system that exploits design patterns for documentation and code recognition. This is the first presentation of an implementation of ordering methodologies.
(2) We show the ease of extension of the system by providing subclasses to incorporate additional algorithms and to incorporate visualization into the system.
(3) We conduct a comparative study using seven existing applications covering a variety of domains. This study provides results about the actual number of cycles in ORDs and the number of classes and edges contained in these cycles. These results show that manual computation of an ordering for class testing is impractical. This is the first such study described in the literature.
(4) We show that using our approach to merging edges together with the *undo* facility of the Command Pattern (Gamma et al., 1995), we can gain an appreciable speedup in the computation of a class order.
(5) We show that in six of our seven applications, it is not possible to break all of the cycles by removing only association and dependency edges.
(6) We show that by including inheritance edges in removal considerations the number of required stubs can be reduced dramatically. The notion of choosing inheritance edges contradicts the advice described in previous research (Briand et al., 2001; Kung et al., 1995; Labiche et al., 2000).

In the next section, we describe an ORD and define the types of edges that represent relationships between classes in the ORD. In Section 3 we review the two methodologies that form the basis of our study and in Section 4 we discuss the implementation of our system. In Section 5 we discuss the particulars of our experiments, including the test suite and the specific questions addressed by the experiments. In Section 6 we describe and analyze the experimental results and provide some insight into the differences between the two methodologies. Finally, in Section 7, we provide some concluding remarks.

2 The ORD and Edge Types

The most common program representation used in class ordering for class-based testing is the Object Relation Diagram (ORD) (Kung et al., 1995; Labiche et al., 2000; Malloy et al., 2003a; Tai and Daniels, 1997). An ORD [1] is a directed graph whose nodes are classes and whose edges represent the

[1] The use of the term ORD is a bit of a misnomer, since the nodes are classes, not objects; however, since the term is used in previous research, we continue its use in this paper.

relationships between the classes. In the sections below we describe the edge types present in the ORD and then give an example.

2.1 Edge type designations

The ORD described in references (Kung et al., 1995; Tai and Daniels, 1997) uses three types of edges, and reference (Labiche et al., 2000) extends the ORD to include a fourth type of edge. However, the focus of our work is the analysis of existing C++ applications which include template functions and classes and nested classes; thus we require six edges in our ORD including the addition of *ownedElement* and *composition* edges.

These edges capture relationships in the ORD between the classes in the program under test and are specified by the syntax and semantics of the data attributes of classes and the parameters or local variables of member functions. The six types of edges are:

- *association*, *dependency*, and *inheritance* as outlined in (Kung et al., 1995; Tai and Daniels, 1997),
- *polymorphic* as described in (Labiche et al., 2000),
- *ownedElement* and *composition*, as given in this paper.

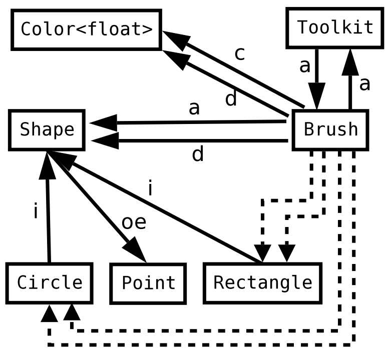
The types of edges, other than polymorphic, are used in UML class diagrams and we base our use of these edges on the UML specification, version 1.5 (OMG Unified Modeling Language Specification, 2003). The *polymorphic* edge is presented in reference (Labiche et al., 2000) as a dynamic edge.

Although there is some controversy about the meanings of various edge types, the differences are in the details and not in regard to the importance of broad classifications. Among the six edge types utilized in this paper, the definitions of *inheritance*, *ownedElement* and *polymorphic* edges are straightforward and are described in the example of Section 2.2. A *composition* edge is used for a class data attribute whose lifetime is bound to the lifetime of the containing object, an *association* edge is used for a class data attribute that is a reference or pointer to another class, and a *dependency* edge is used for a parameter or local variable of a member function.

2.2 A sample ORD

Figure 1(i) lists a C++ program and illustrates the corresponding ORD for the program. Most details of the classes are elided for brevity; however, the information provided is sufficient to demonstrate each of the six edge types

```
( 1) template < typename T >
( 2) class Color {
( 3)   T  r, g, b, a;
( 4) };
( 5)
( 6) class Shape {
( 7) protected:
( 8)   class Point { };
( 9) };
(10)
(11) class Circle    : public Shape { };
(12) class Rectangle : public Shape { };
(13)
(14) class Toolkit;
(15)
(16) class Brush {
(17) public:
(18)   void setColor(
          const Color<float>& color);
(19)   void setShape(const Shape* shape);
(20) private:
(21)   Toolkit      *parent;
(22)   Color<float>  color;
(23)   Shape         *shape;
(24) };
(25)
(26) class Toolkit {
(27)   Brush *brushes[10];
(28) };
```



(i) (ii)

Fig. 1. *Sample C++ program and corresponding ORD.* This figure lists a sample C++ program and a corresponding ORD for the program. This ORD illustrates the six types of edges included in our model: the four dashed lines in the graph are *polymorphic* edges, and the other edges are labeled appropriately.

in our ORD. A template class, Color, is listed on lines 1 through 4 of Figure 1(i). A template class has no direct representation in the ORD, due to the fact that a template class represents only a partial specification of a class whose arguments can be either primitive types or other classes. Before a template class can be executed (and subsequently tested) *template instantiation* must first be performed by supplying actual arguments for the formal template parameters. An instance of Color, instantiated with the primitive type **float** appears on the right of Figure 1(ii).

Lines 6 through 9 of Figure 1(i) list class Shape, which contains a nested class Point. An *ownedElement* edge from Shape to Point is generated by this relationship, and is illustrated to the left of center in Figure 1(ii).

Lines 11 and 12 of Figure 1(i) list classes Circle and Rectangle, both derived publicly from base class Shape. These derivations generate *inheritance* relationships illustrated in Figure 1(ii) by the edges from Circle to Shape and Rectangle to Shape, both of which are labeled *inheritance*.

Line 14 of Figure 1(i) lists a forward declaration of class Toolkit. Lines 16 through 24 of Figure 1(i) list class Brush, which contains two public methods and three data attributes. Formal parameters and local variables of methods form *dependency* relationships between classes; therefore, there are two *dependency* edges generated by the methods of class Brush. A *dependency* edge from Brush to Color is generated by parameter color of method setColor (line 18 of Figure 1(i)). The second *dependency* edge, this time from Brush to Shape, is generated by parameter shape of method setShape (line 19 of Figure 1(i)). Data attributes form either *association* or *composition* relationships between classes. Class Brush contains three data attributes, the first of which is parent, a pointer to Toolkit. An *association* edge from Brush to Toolkit is generated by this data attribute since Brush has a relationship with Toolkit, but the relationship can be severed or transferred before destruction of the Brush object. The *association* relationship is not as strong as the *composition* relationship, which cannot be broken until the object owning the data is destroyed. A *composition* edge from Brush to Color<**float**> is generated by the second data attribute of class Brush, color. The final data attribute of Brush is a pointer to Shape, generating an *association* edge between Brush and Shape.

When a class has an *association* edge to the base class of an inheritance hierarchy, a *polymorphic* edge is generated. A *dependency* edge may also generate a *polymorphic* edge if the parameter responsible for generating the *dependency* relationship is a pointer or a reference to a base class. Polymorphic edges were first described in (Labiche et al., 2000), and referred to as dynamic edges; however, these edges can be determined statically, so we refer to them as *polymorphic* edges. As described in the previous paragraph, class Brush has a *dependency* relationship with class Shape, a base class. This *dependency* edge from Brush to Shape generates *polymorphic* edges from Brush to each of the classes derived from Shape, because the parameter is a pointer. Also mentioned above is the *association* edge from Brush to Shape that is generated by data attribute shape. Again, *polymorphic* edges from Brush to each class derived from Shape are generated by this *association* relationship. The four *polymorphic* edges are illustrated at the bottom of Figure 1(ii) as unlabeled dashed lines from Brush to Circle and Brush to Rectangle.

Lastly, class Toolkit is listed on lines 26 through 28 of Figure 1(i). Class Toolkit contains a data attribute brushes which is an array of pointers to Brush. This data attribute generates an *association* edge from Toolkit to Brush, and creates a cycle in this example ORD. This cycle is illustrated in the upper right of Figure 1(ii).

3 Class Ordering Methodologies

In this paper we provide a comparative study of two advanced methodologies for class ordering, namely, an edge based approach by Briand, et al. (Briand et al., 2001), and the Class Ordering System (COS) introduced here. In the remainder of this section we provide detailed descriptions of these two methodologies. In the case of Briand, our description is equivalent to that provided in (Briand et al., 2001), although we have rewritten the specifics to provide a uniform framework under which the methodology can be compared with our COS approach. At the end of the section we describe some of the other related work.

The approach of Briand and our COS can both be described based on a framework of three stages:

- **Stage 1**: Build an ORD. That is, construct a multigraph G = (V, E), where V is a set of nodes representing classes, and E is a set of edges representing the relationships between the classes. A multigraph may contain multiple edges between any particular pair of nodes. For each of the C++ applications in the test suite utilized in this paper, the associated ORD was obtained by reverse engineering the source code of the application using the Clouseau API (Matzko et al., 2002).
- **Stage 2**: Remove edges from G so that all of its cycles are broken. Let G' be the resulting acyclic ORD.
- **Stage 3**: Determine a class ordering for testing by ordering the classes in G' in reverse topological order.

Note that if an edge (x,y) is removed in stage 2, then the testing of class x will depend on a stub associated with class y. Both of the methods (Briand et al., 2001) and (Malloy et al., 2003a) are aimed at minimizing the number of stubs that must be written. In the sections below we provide details for the two methods in regard to assumptions and implementation of stages 1 and 2 of the above framework.

```
while there is an SCC with more than one node do
  Choose an SCC, $s$, with more than one node
  for each association/dependency edge (x, y) in $s$,
    Assign a weight equal to indegree(x)*outdegree(y),
    where these degrees are with respect to nodes in
    this SCC only. Here, the node degrees refer to
    all types of edges, and not just to
    association/dependency edges.
  Remove from $s$ an edge of maximum weight.
```

Fig. 2. *The Briand Algorithm.*

3.1 The Briand approach

The methodology of Briand (Briand et al., 2001) implements stage 1 in a straight-forward fashion. In stage 2, cycles are broken by partitioning the ORD into strongly connected components (SCCs) and then selecting edges within each SCC for removal based on an edge weight meant to reflect the number of cycles containing that edge. The specifics are illustrated in Figure 2.

Recall that in our classification, there are six possible types of edges between C++ classes. As evident from the pseudocode in Figure 2, Briand (Briand et al., 2001) focused exclusively on *association* and *dependency* edges, and as such did not consider *polymorphic*, *inheritance*, *composition* or *ownedElement* edges. In implementing the Briand methodology in conjunction with class diagrams utilizing all six edge types, we need to modify the approach since it is not always possible to remove all cycles by only removing *association* and *dependency* edges. In fact, among the seven programs in our test suite, there is only one for which all of the cycles can be removed using only *association* and *dependency* edges. It *is* the case that all of the cycles in all of the test cases in our test suite can be broken by removing *polymorphic* edges in addition to *association* and *dependency* edges. Thus, we implement two modifications to the Briand approach:

(1) In stage 2, *polymorphic*, *association* and *dependency* edges are all considered, with no preference being given to any particular edge type. This variant of (Briand et al., 2001) will be referred to as **Briand-A**.
(2) In stage 2, once an SCC is chosen, if that SCC contains an *association* or *dependency* edge, then the steps of the algorithm proceed as outlined in Figure 2. If however that SCC does not contain an *association* or *dependency* edge, then the second and third steps of the Briand while loop are executed using *polymorphic* edge in place of *association/dependency* edge. This variant of (Briand et al., 2001) will be referred to as **Briand-B**, and reflects the argument that it may be better to remove (i.e. easier

to stub) *association* and *dependency* edges.

Since *association*, *dependency* and (with the above addition) *polymorphic* edges are the primary focus of the Briand approach, we find it convenient throughout the remainder of this paper to use the term *ADP edges* when referring to these three types of edges.

3.2 The Class Ordering System (COS)

The Class Ordering System (COS) described here utilizes a generalized approach to class ordering that reflects differences in the suitability of various edge types for stubbing. The following two subsections describe the COS approach in the context of the three general stages outlined earlier.

3.2.1 The COS cost model applied to the ORD

In the COS methodology, the standard construction of the ORD in stage 1 is augmented using a *cost model* $C = <\mathcal{W}, f(e), w(m_{x,y})>$, which is a 3-tuple consisting of $\mathcal{W}$, a set of weight assignments and functions $f(e)$ and $w(m_{x,y})$ defined as follows:

$$\mathcal{W} = \{w_1, w_2, w_3, w_4, w_5, w_6\} \tag{1}$$

$$f \; : \; E \rightarrow \mathcal{W} \tag{2}$$

$$for\ a\ given\ x, y \in V, \;\; m_{x,y} \; = \; \{(x, y) \in E\} \tag{3}$$

$$w(m_{x,y}) = \sum_{e \in m_{x,y}} f(e) \tag{4}$$

Equation (1) is a set of weight assignments for the six edge types of *inheritance*, *association*, *composition*, *dependency*, *polymorphic* and *ownedElement* edges. Equation (2) defines a total function f as a mapping from the set of edges, E, to the set of weights $\mathcal{W}$, so that for edge e, $f(e)$ is the weight assignment for that edge. Equation (3) defines a *merged* edge $m_{x,y}$ as a set of edges represented as ordered pairs (x, y), where each edge in the set has the same source class and the same destination class. Equation (4) defines $w(m_{x,y})$, a function w that computes the weight of a *merged* edge $m_{x,y}$ as the sum of the weights of the individual edges in the set $m_{x,y}$.

Applying this cost model to the standard ORD results in the merging of multi-edges and the assignment of a weight to each resulting edge as defined by the cost model. We refer to this modified ORD as the COS ORD and it is the output of stage 1 using the COS methodology.

```
while there is an SCC with more than one node do
    Choose an SCC, $s$, with more than one node;
    Remove from $s$ an edge of minimum weight.
```

Fig. 3. *The COS Algorithm.*

3.2.2 Stage 2 of the COS methodology

The COS methodology implements stage 2 similarly to that of (Briand et al., 2001), albeit as applied to the COS ORD with *merged* and weighted edges. Specifically, cycles are broken by partitioning that ORD into strongly connected components and then selecting edges within each SCC for removal based on the weight of an edge. The specifics are illustrated in Figure 3.

A critical question is how to choose among edges of equal weight. Having such edges is much more likely in this methodology than in (Briand et al., 2001), since the weights are based on edge types and not on in and out degrees. For instance, all *association* edges will have the same weight (unless they have been *merged* with other parallel edges). Consequently, we investigate three versions of COS based on choosing between edges of equal weight:

(1) Among all edges of minimum weight, remove the one having the smallest index source node. This version is easy to implement, but skews edge selection toward edges incident to low index nodes. This variant will be referred to as **COS-A**.
(2) Among all edges of minimum weight, select an edge at random and remove it. The goal is to spread the selection of removed edges among all of the nodes in the ORD. This variant will be referred to as **COS-R**.
(3) Assign each edge a *secondary weight* based on the weighting scheme utilized by Briand (Briand et al., 2001). Namely, the secondary weight of edge (x,y) is equal to the indegree(x)*outdegree(y), where the in and out degrees take into account only the edges in this SCC. Then, in removing an edge of minimum weight, if there are several such edges of minimum weight, choose one with largest secondary weight. This variant will be referred to as **COS-B**.

3.3 Other related work

In (Daniels and Tai, 1999), Tai and Daniels study a node based approach to cycle removal in an ORD. Their method is similar to (Briand et al., 2001) and (Malloy et al., 2003a) in that they partition the ORD into SCCs (though their particular partitioning differs a bit). Focusing on *association* and *dependency* edges, they remove *nodes* from the ORD until all cycles are removed. Here, the classes associated with removed nodes are stubbed. It is argued in (Briand et al., 2001) that the stubbing of nodes is considerably more expensive than stubbing based on individual edges.

In (Le Traon et al., 2000), Le Traon et al. use an adaptation of Tarjan's algorithm (Tarjan, 1972) to find strongly connected components in a graph. While traversing their graph, dependencies are labeled according to a classification scheme where one type of classification is a *frond*, which indicates an edge from a node to an ancestor of the node. Cycles are broken by removing edges of highest weight, where *weight* is defined as the sum of the incoming and outgoing frond dependencies for a given class in the graph. Thus, the notion of weight is defined on classes rather than edges and they do not include the edge types that subsequent researchers have incorporated into an ORD (Kung et al., 1995; Labiche et al., 2000; Malloy et al., 2003a; Tai and Daniels, 1997). Briand et al. (Briand et al., 2001) showed that their approach performed consistently better than that of Le Traon et al. (Le Traon et al., 2000).

Some of the pioneering work on class ordering was accomplished by Kung et al. (Kung et al., 1995). However, that work is subsumed by Briand et al. (Briand et al., 2001); hence, we do not review (Kung et al., 1995) in this paper.

A paper describing an approach for maintaining the accuracy and consistency of software development artifacts is presented in reference Lu and Chu (2003). The work uses XML to provide interoperability among the development artifacts.

4 Design of the System

A preliminary version of our class ordering system, which served as the basis for the current implementation, was presented in (Malloy et al., 2003a). The design of the preliminary version was resistant to extension and modification and only the **COS-A** methodology was included in the implementation. We refactored and augmented the preliminary version to include several design patterns (Gamma et al., 1995), which has resulted in an efficient and extensible system that is configurable and incorporates new features.

The following subsections present the design and implementation of our configurable and extensible *Class Ordering System*, COS. We present an overview of the operations of COS in Section 4.1 and review the important classes and design patterns in Section 4.2. In Section 4.3 we illustrate the extensibility of our system to accommodate visualization and to accommodate node-based removal methodologies.

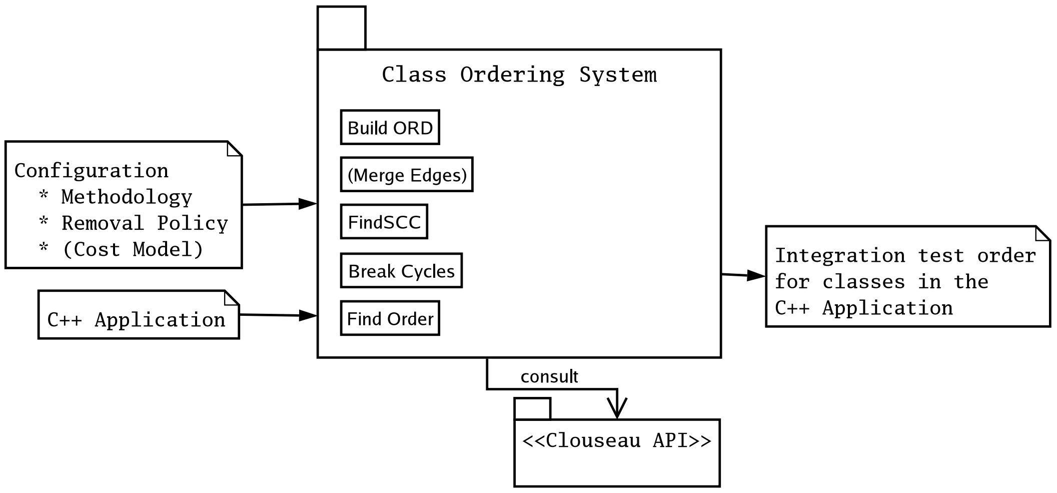


Fig. 4. *Overview of our Class Ordering System.*

4.1 Overview of the System

Figure 4 provides an overview of our *Class Ordering System*. Inputs to the system are a configuration file and a C++ application on which class-based testing is to be performed. The configuration file specifies the desired methodology and, if the specified methodology is a variant of the COS methodology, then one or more cost models are also included.

Depending on the methodology under consideration, we compute an ordering of classes for testing in four or five steps: each of the Briand variants requires four steps and each of the COS variants requires five steps. The first step entails the construction of an ORD for the application using the six edge type designations that describe the relationships between the classes in the application. The next step, which is only performed in the COS variants, is to merge edges in the ORD as described in Equation (3) of Section 3.2.1.

After ORD construction (and the merging of edges for COS variants) is complete, the nodes of the ORD are partitioned into strongly connected components, SCCs, using a depth first search as described in (Aho et al., 1974). In the next step edges are removed from the SCCs and the SCCs are recomputed until no cycles remain in the ORD. In the final step a reverse topological sort of the nodes in the modified ORD yields an integration order for class-based testing of the system under consideration.

To build an ORD for the C++ application under test, the source code for the application is parsed, and variable and type information is extracted from the application. To parse our application we use *keystone*, an ISO conformant parser and front-end (Malloy et al., 2003b) for the C++ language (ISO/IEC JTC 1, 1998; Stroustrup, 1997). *Keystone* includes the *Clouseau* Application

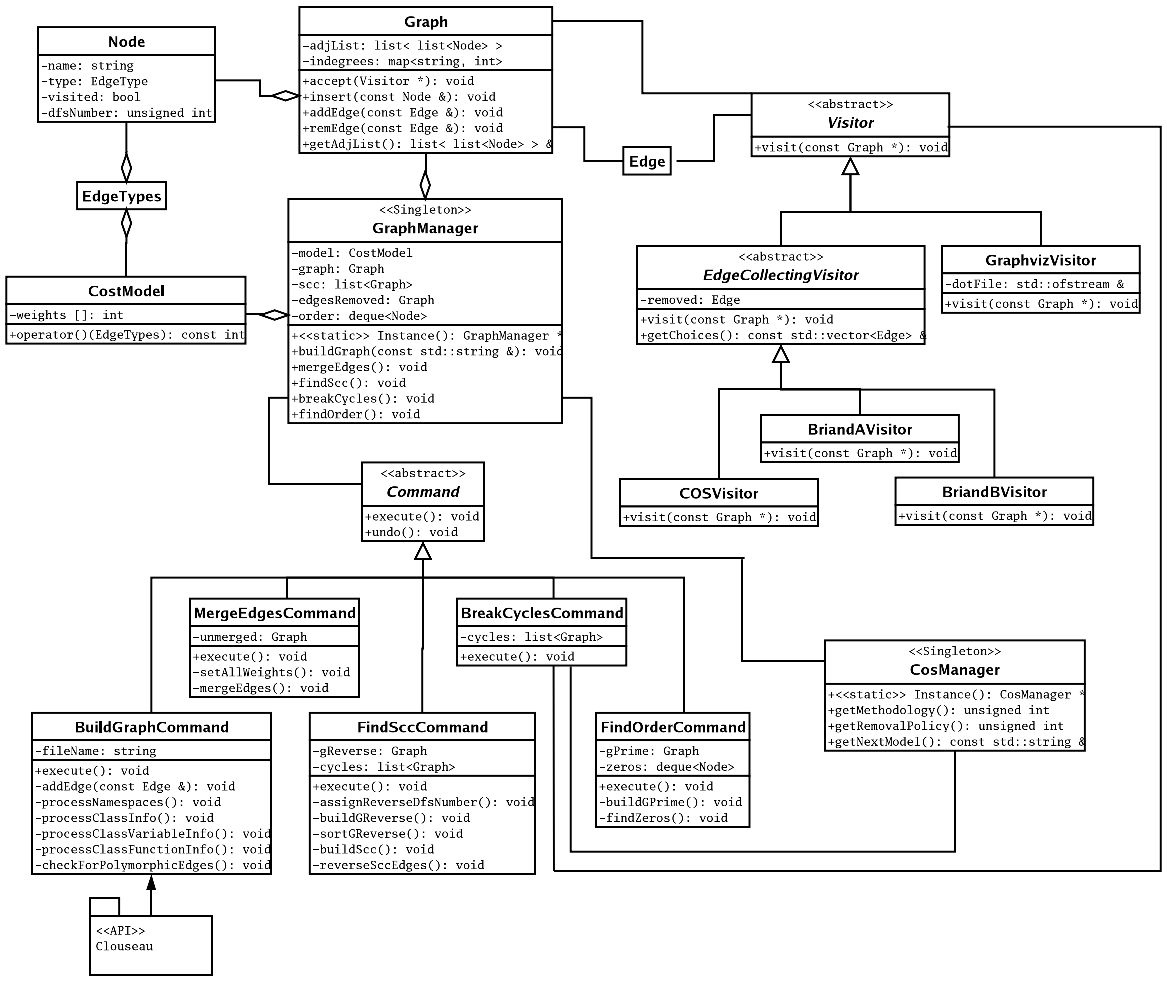


Fig. 5. *COS Class Diagram.* This figure contains a UML class diagram illustrating the important classes and relationships in our Class Ordering System, COS.

Programmer's Interface (Matzko et al., 2002), API, which provides the necessary variable and type information required to build an ORD for the application.

4.2 The Important Classes in the System

Figure 5 contains a UML class diagram illustrating the classes and important relationships in our Class Ordering System (COS); some relationships have been elided from the diagram for readability. Class CosManager, shown in the lower right of the figure, illustrates the Singleton design pattern and serves as an access point to the system configuration. The three classes in the upper left of the figure, Graph, Node, and EdgeTypes, are used to build the nodes and edges in an ORD. Class EdgeTypes represents the six edge types in our ORD, described in Section 2.1. The CostModel class, also shown in the upper left of the figure, encapsulates the mapping of edge types to weights as defined in Equation (2) of Section 3.2.1. The class in the middle of the figure, GraphManager, is another Singleton that choreographs the steps of the selected methodology. The six classes in the inheritance hierarchy at the bottom of the figure, Command, BuildGraphCommand, MergeEdgesCommand, FindSccCommand, BreakCyclesCommand and FindOrderCommand, illustrate the Command design pattern and encapsulate the steps of the algorithm for each methodology. The Clouseau API is shown in the lower left of the figure. The inheritance hierarchy in the upper right of the figure including Visitor, EdgeCollectingVisitor, COSVisitor, BriandAVisitor, BriandBVisitor and GraphvizVisitor, illustrate the Visitor design pattern that we use to gather information about the ORD from class Graph.

In the following sections we further describe the design of COS using the design patterns as a focal point.

4.2.1 The Singleton Pattern

The Singleton design pattern is used to provide a global access point to a single instance of a class (Gamma et al., 1995). There are two singletons in our design: CosManager, at the lower right of Figure 5, and GraphManager, in the middle of Figure 5; each singleton has a static method, Instance, which returns a pointer to the respective instantiated singleton. The CosManager class is responsible for reading a configuration file and is queried by the GraphManager and BreakCyclesCommand classes. The use of the Singleton pattern for CosManager allows information such as the currently selected methodology and the name of the file containing the active cost model to be accessible to the other objects in the system. The GraphManager class is the primary choreographer of the system: GraphManager contains the ORD, provides the current cost model (if needed), and invokes the commands that perform the steps of the selected methodology. In the case of GraphManager, use of the Singleton allows command invocation without passing data to the individual commands.

4.2.2 The Command Pattern

The Command design pattern provides a common interface for transactions in a system by encapsulating a series of actions, such as the steps of an algorithm, into a class (Gamma et al., 1995). The Command pattern facilitates extension of the system to accommodate new commands through subclassing. By encapsulating the actions in a class, objects of the class can be passed as parameters or stored to permit efficient reversal or undoing of the actions of the object. The Command pattern usually takes the form of an abstract base class, Command, with a pure virtual method, execute(), that provides an interface for executing operations. Concrete Command subclasses provide an implementation of execute() that contains the actions of the particular command.

The inheritance hierarchy in the lower left of Figure 5 illustrates a Command pattern consisting of base class Command and derived classes BuildGraphCommand, MergeEdgesCommand, FindSccCommand, BreakCyclesCommand and FindOrderCommand. The derived classes encapsulate the steps performed by each of the ordering methodologies. Instantiation and execution of the different commands is controlled by the GraphManager class, which discerns the appropriate commands to run by querying the CosManager. This use of the Command pattern allows both selective execution of existing commands and the introduction of new commands with minimal change to the system.

Our system exploits the undo capability of the Command pattern to improve the efficiency of the system. For example, if a variant of the COS methodology is under investigation, the system can be configured to perform several class orderings with a series of cost models. The efficiency of this process is improved by using an undo method in the MergeEdgesCommand. The undo of the edge merging process allows the internal state of the system to return to a point where the cost model has yet to be applied to the ORD. The time savings here are significant because rebuilding of the ORD is no longer required. A comparison of system running times for several cost models, with and without the undo capability, is presented in Section 6.5.

4.2.3 The Visitor Pattern

The Visitor design pattern permits encapsulation of operations that are to be performed on the elements of a data structure (Gamma et al., 1995). The addition of new operations and modification of the existing operations is performed without changing the classes that represent the elements. This design pattern is especially useful when the representation of the elements of the data structure is stable and allows the classes that represent the elements of a data structure to be decoupled from the algorithms that operate on the elements. This decoupling prevents the classes representing the elements from becoming polluted with the algorithms that operate on the elements. Finally, the visitor pattern facilitates extension of an existing system by permitting the addition of new operations and algorithms without modifying the classes that represent the elements of a data structure. To incorporate new functionality into an existing structure, each element of the structure accepts a visitor, which sends a message to the visitor that includes the element class. Using this double dispatch, the visitor then executes its algorithm on the elements of the structure.

Our Visitor hierarchy is shown in the upper right of Figure 5 consisting of base class Visitor and five derived classes EdgeCollectingVisitor, COSVisitor, BriandAVisitor, BriandBVisitor and GraphvizVisitor. Each of the visitor classes listed in the figure contains a method, visit(const Graph *), to allow visitation of the ORD represented by class Graph (upper left of the figure). The Graph class contains a corresponding accept method, accept(Visitor *), which calls the visit method passing an instance of the visited class through the this pointer (Gamma et al., 1995).

Our Visitor hierarchy incorporates the features, described above, into the COS. In particular, the three edge collecting visitors encapsulate the collection of edges which are candidates for removal during cycle breaking, decouple the cycle breaking methodologies from the Graph class and enable extension to other edge-based cycle breaking methodologies through subclassing. In Section 4.3.2 we describe an approach to extend the Visitor hierarchy to accommodate node-based removal methodologies. The GraphvizVisitor class illustrates a different kind of extension that enables us to visualize our ORD and the cycle breaking process, facilitating comprehension, debugging and validation of the ORD and the cycle breaking process. We describe the actions and output of the GraphvizVisitor in Section 4.3.1.

There are two phases to the cycle breaking process: the first phase entails an edge collecting strategy and the second phase entails an edge removal policy. The classes in our EdgeCollectingVisitor subhierarchy implement an edge collecting strategy for each methodology in the system. For example, the common edge collecting strategy of the COS variants is implemented in class COSVisitor. The visit method of COSVisitor collects all edges of lowest weight from the SCC under consideration. Note that each variant selects an edge to be removed from among the collected edges in a different manner. Alternatively, the distinguishing characteristic of the Briand variants is the edge collecting strategy rather than the removal policy; therefore, the Briand variants each require their own visitor class.

4.3 Extensibility of the System

In this section we illustrate the extensibility of our system by presenting an extension to accommodate visualization and an extension to accommodate node-based removal methodologies.

4.3.1 Extension to Accommodate Visualization

During early development of COS we performed debugging and correctness evaluation of the ORD and the cycle breaking process using a textual rep-

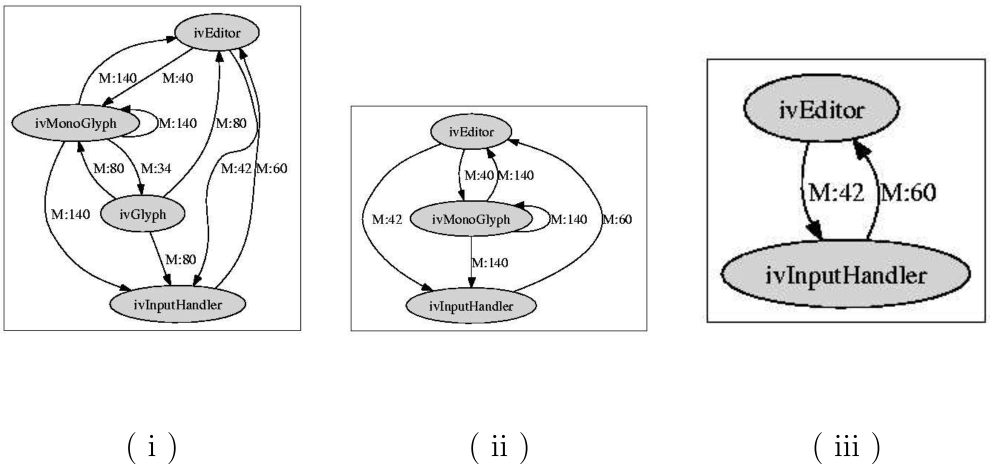


Fig. 6. *Visualization of the cycle breaking process.* This figure illustrates a progression of edge and node removals as the COS is used to break cycles for the IV-Graphdraw test case using the **COS-A** methodology. The placement of nodes and edges in this figure is computed automatically by our `GraphvizVisitor` together with the *dot* tool in the graphviz package: this sequence of graphs illustrates the excellent placement strategy of the *dot* tool and facilitates comprehension, debugging and validation of the ORD and edge removal process.

---

resentation of the ORD. However, this method of evaluation proved to be tedious and error-prone. To address this difficulty of validating the COS output, we extended the system to include visualization. The visitor hierarchy, described in Section 4.2.3, facilitated this extension through the addition of a GraphvizVisitor class as a subclass of the Visitor class.

Visualization for our system is accomplished by producing graphical output using the *dot* tool, included in the Graphviz package (AT&T Labs, 2005). The class GraphvizVisitor builds a file containing a *dot* language representation of the graphs in our system, which are encapsulated by class Graph. By modifying the configuration file, the system can be directed to produce a *dot* representation for any of the following graphs: the ORD, the ORD with edges merged, each individual SCC, and the cycle-free graph upon which the reverse topological sort is applied.

Figure 6 illustrates a progression of edge removals that occur when the COS is used to break cycles for the IV-Graphdraw (Vlissides and Linton, 2002) test case using the **COS-A** methodology. Each graph in Figure 6 is an ORD representing an SCC: the ellipses are nodes that represent classes and the edges represent relationships between the classes. Each node is labeled with the name of the class that it represents and the edges are labeled with the edge type and the associated weight of the edge. All of the edges in Figure 6 are *merged* edges so each edge is labeled M, for merged edge. For example, in Figure 6(i), the edge connecting ivMonoGlyph and ivGlyph is labeled M:34, indicating that it is a merged edge with a weight of 34, the lowest weighted edge in Figure 6.

To visualize the cycle breaking process for the **COS-A** methodology applied to the IV-Graphdraw test case, Figure 6(i) shows a starting point with four classes and nine edges. The edge connecting ivMonoGlyph and ivGlyph is labeled M:34, the edge with lowest weight and this edge is chosen for removal, also removing the node representing class ivGlyph from the graph since it is no longer involved in a cycle. Figure 6(ii) illustrates the next step in the cycle breaking process with three classes and six edges. The edge connecting classes ivEditor and ivMonoGlyph has weight 40, the edge with lowest weight in the graph of Figure 6(ii); this edge together with class ivMonoGlyph is removed from the graph[2]. Figure 6(iii) illustrates the next phase in the cycle breaking process, with two classes and two edges remaining in the SCC. The next edge to be removed is the one from class ivEditor to ivInputHandler, representing the final edge removal from the original SCC presented in this figure.

The placement of all nodes and edges in Figure 6 is computed automatically by the *dot* tool; the sequence of graphs illustrates the excellent placement strategy of *dot*. We found the visualization of the cycle breaking process to be easier to debug and validate than the corresponding text based representation. Appendix A contains a figure illustrating a visualization overview of the ORD for the vkey test case (Wampler, 2001). The ORD in this figure graphically illustrates the tight coupling among the classes in the vkey test case. The SCC shown in the middle of the figure consists of 27 classes; the clusters of classes shown in the upper left and the lower right of the figure are not SCCs. We provide further discussion of the vkey test case in Section 6. However, the figure in the appendix, like those in Figure 6, was also drawn automatically and further illustrates the advantage of the visualization capability of COS and the excellent node and edge placement strategy of the *dot* tool.

4.3.2 Extension to Accommodate Node-Based Removal Methodologies: Tai/-Daniels

In this section we further illustrate the extensibility of COS by presenting a strategy to accommodate node-based removal methodologies into the system. The only node-based removal methodology presented in the literature is that of Tai and Daniels (Tai and Daniels, 1997). This early work on class ordering, like the methodology of Briand (Briand et al., 2001), fails to include polymorphic edges in their graph representation and in their removal considerations. By not including polymorphic edges in their graph and removal strategy, impor-

[2] When the SCCs are recomputed, the self-loop on node ivMonoGlyph represents a cycle that must also be broken; otherwise, we cannot compute a reverse topological sort of the nodes in the reduced ORD.

tant class dependencies are also not represented. Nevertheless, the approach is interesting both in its novelty and in the fact that, ultimately, it is classes that are stubbed. To accommodate the approach of Tai and Daniels, we must modify their approach to include polymorphic edges in their graph representation and in their cycle breaking considerations.

Moreover, to accommodate the approach of Tai and Daniels, we extend the COS in the obvious way: the addition of a new command in the Command hierarchy and the addition of a new visitor in the Visitor hierarchy. Like the COS variants, the methodology of Tai and Daniels entails five steps; however, instead of executing the merge edges command, the Tai and Daniels methodology executes a new command that assigns level numbers. Therefore, the Command hierarchy is extended to include class LevelAdgCommand, a command to encapsulate algorithm Level_ADG as presented in (Tai and Daniels, 1997).

The second extension to COS is a subclass of Visitor, class TaiDanielsVisitor. As described in Section 4.2.3, the edge collecting strategy of each methodology is implemented by a graph visitor. The TaiDanielsVisitor determines the node to be removed under the Tai and Daniels methodology, and then collects all edges originating or terminating at that node. The final addition is a private method of BreakCyclesCommand implementing the straightforward policy of removing all collected edges.

5 A Comparative Study

In this section we provide specifics of our experimental study undertaken here, beginning with a description of our test suite and following with a discussion of stub creation as it relates to edge type.

5.1 The test suite

Our experiments use a suite of seven test cases that we label as Adol-C, Class Ordering System (COS), matrix, vkey, Edraw, Graphdraw and Drawserv. In our study, a *test case* is a program that we use as input to the COS and Briand systems. The test cases were chosen for their range and variety of application. Specifically, test case Adol-C is a package for automatic differentiation of algorithms (A. Griewank and O. Vogel, 2003) and COS is the preliminary version of the *Class Ordering System* described in (Malloy et al., 2003a). The matrix test case is an extended precision matrix application that uses *NTL*, a high performance portable CPP number theory library (Shoup, 2002). vkey is a GUI

Table 1. *Test Suite and Statistics about Strongly Connected Components (SCC). For the test cases there were very few (no more than 3) SCCs containing more than one class. On the other hand, those few SCCs can be very large, containing nearly 5000 edges in the case case of the* Drawserv *test case. The* matrix *test case, listed on line 3, did not contain any cycles, although there are self loops for some of the classes; some classes contain as many as four (4) of these self loop edges, as illustrated on the third row, last column of data in this table.*

Test case	**LOC**	**Classes**	**Edges**	**SCC with 1 class**	**SCC with >1 class**	**Largest SCC (classes)**	**Largest SCC (Edges)**
Adol-C	699	16	111	7	2	5	63
COS	1304	43	138	40	1	3	11
matrix	4944	50	164	50	0	1	4
vkey	8588	46	226	27	1	19	143
Edraw	832	44	252	30	2	12	117
Graphdraw	4354	151	1340	106	3	39	673
Drawserv	5687	236	6460	118	3	110	4722

application that uses the *V GUI* library (Wampler, 2001), a multi-platform CPP graph framework for GUI applications. The final three test cases, Edraw, Graphdraw and Drawserv, are GUI applications generated from the *Tools* drawing application (Vlissides and Linton, 2002), a suite of free XWindows drawing editors for Postscript, TeX and web graphics production.

Table 5.1 lists summary information about the ORDs and the cycles in the ORDs for the test cases in our study. The first column of data lists the lines of code in each test case, the next two columns of data list information about the respective ORD and the final four columns list information about the strongly connected components (SCCs) present in the ORD. It is interesting to note that in all test cases there were very few (no more than 3) SCCs containing more than one class. On the other hand, those few SCCs can be *very* large, containing nearly 5000 edges in the Drawserv test case.

The matrix test case, listed on line 3 of Table 5.1, did not contain any SCCs with more than one class; thus, there are no cycles in this test case. Nevertheless, for some of the *singleton* SCCs with only one class, there are self

Table 2. *Edge types in each test case.*

Number of:	Adol-C	COS	matrix	vkey	Edraw	Graphdr	Drawserv
Inheritance edges	7	11	0	14	10	42	110
Association edges	1	7	18	33	8	93	167
Composition edges	0	39	27	8	4	7	9
Dependence edges	38	79	119	88	150	527	1387
Polymorphic edges	65	2	0	83	80	671	4787
Owned Element edges	0	0	0	0	0	0	0

loops for the class so that some classes had as many as four (4) edges, as illustrated on the third row, last column of data in the table. Self loops occur frequently in real applications, generated by self-referential functions whose parameter(s) are instances of the class. Examples of self-referential functions are *copy constructors*, *overloaded assignment operators* and other functions for binary operators.

Table 5.1 lists the number of edges of each type in each test case. Note that the vast majority of edges are ADP edges (97% overall). Nonetheless, as noted earlier, in six out of seven test cases (matrix being the exception) it is not possible to break all of the cycles using just association and dependency edges. One fact that does not appear in the results shown in Table 5.1 concerns composition edges, namely that although our test cases contain some composition edges, *none* of those composition edges is located in a cycle.

All of our experiments were executed on a workstation equipped with two *Intel*© *Pentium*© *III* 1.0 GHz processors, 1 GB of SDRAM, and running the Suse Linux 9.0 operating system.

5.2 The difficulty of stubbing vs edge type

There is considerable uncertainty about the cost of creating stubs relative to various types of edges. Further, when considering the stubbing necessary to accomplish class-based testing of a large program, there may be a trade-off between the effort required to stub an individual method and the total effort taken over all stubs needed for the testing. For instance, it might be preferable to create three *difficult* stubs rather than 40 *easy* stubs.

Thus, in this study we investigate the cost of stubbing using two measures of the goodness of the stubbing:

- **The number of stubs**: What is the total number of stubs that are created (i.e. the number of edges removed), and how does this number of stubs compare with stubbing based exclusively on ADP edges? For this experiment we run Briand-A, Briand-B, and use two COS cost models with the three versions of COS. The first cost model uses a cost of 2 for ADP edges, and a cost of 9999 for all other edges. The effect of using these costs is to disallow the stubbing of non-ADP edges. This will allow a direct comparison of the Briand and COS approaches, since only ADP edges will be removed. The second COS cost model uses a cost of 2 for all edges. Since all edge costs are identical, this will allow a direct evaluation of the advantage (if any), in terms of the total number of stubs, of stubbing all types of edges versus just stubbing ADP edges.
- **The overall cost of stubbing**: Since the ultimate goal of this work is to reduce the total cost of class-based testing, an alternative measure of stubbing is to reflect both the number of stubs and the cost of producing those stubs. From (Briand et al., 2001; Kung et al., 1995) there is general agreement that the stubbing of association, dependency and polymorphic edges can be relatively straight-forward, and that such ADP edges are the cheapest to stub among all edge types. Further, it seems that stubbing composition[3] edges is considered to be more difficult than for ADP edges. The biggest uncertainty is in regard to the stubbing of inheritance edges. On the one hand, it is stated in (Briand et al., 2001) that the stubbing of inheritance edges is not "economically viable". On the other hand, recent work (Malloy et al., 2003a; Lloyd and Malloy, 2005) suggests that stubbing of inheritance edges may be reasonable in some circumstances. Thus, in enumerating appropriate COS cost models based on the cost of creating stubs for different types of edges we have the following relationships[4]: cost(dependency) = cost(association) = cost(polymorphic) < cost(composition) = cost(ownedElement) << cost(inheritance). The overall cost of stubbing is then the sum over all removed edges of the cost associated with edges of that type.

  Clearly, the overall cost of stubbing obtained from experiments will be highly dependent on the specific COS cost model that is used. Since there is no way to determine precise values, we consider five COS cost models, each of which conforms to the cost relationships given above, but which differ in the exact cost values that are used for inheritance edges in relation to the cost of ADP and composition edges. In each of those five cost models, ADP

[3] In (Briand et al., 2001), composition edges are denoted as aggregation edges, which are distinguished from *simple aggregations* that are labeled there as a type of association.

[4] For completeness we include ownedElement in these costs, although no such edges actually appear in any of our test cases.

edges are assigned a cost of 2, and composition and ownedElement edges are assigned a cost of 4. The cost of inheritance edges is set to be 80, 40, 20, 10 and 5 respectively, for the five cost models. The variation in the cost assigned to inheritance edges allows us to explore general trends in regard to overall costs, rather than specifics related to a given inheritance cost.

6 The Implementation: Results and Discussion

In this section we describe and analyze the results of our experiments. Throughout this section we provide tables showing the number of edges removed by the variants of the Briand and COS algorithms. For both methodologies, the number of edges shown is the number of *individual* edges. This is straight-forward for Briand, but requires some explanation in the case of COS which utilizes the COS ORD with merged edges. For the variants of COS, the number of individual edges that are removed is computed by summing, over all removed merged edges, the number of individual edges included in each merged edge. For instance, if one inheritance, one polymorphic and two dependency edges from the standard ORD were merged to create a merged edge e, then if e is removed by COS, the result is that 4 stubs will need to be created - one for each of the individual edges that constitute that merged edge. Although it is natural to think that perhaps these four edges could be handled using a single stub, that is not usually the case. For instance, the two dependency edges may reflect calls of different methods in the supplier class. Thus, with the exception of Section 6.3 where we address an issue for which the number of merged edges is a consideration, all of our results refer to individual edges (and hence to the actual number of stubs that need to be created).

In the subsections that follow we address four specific issues.

6.1 Minimizing the number of stubs

Table 6 provides a direct comparison of the Briand and COS approaches to edge selection for breaking cycles in an ORD, and hence to stubbing. There are three aspects to that comparison.

First, recall that there are two versions of Briand based on how that methodology is extended to apply to polymorphic edges. In all test cases the one pass method of Briand-A, which treats polymorphic, association and dependency edges as equally desirable for breaking cycles, is equal or superior to Briand-B, which only breaks polymorphic edges if no association or dependency edges are available. However, the differences between the two methods are relatively

Table 3. *Number of removed edges: Briand and COS.* The cost model 6-tuple gives the costs (in order) for: inheritance, association, composition, dependency, polymorphic, and ownedElement edges. There were 88 edges removed for the matrix test case for all four models; these edges are all self loops induced by functions such as *copy constructors* and *overloaded assignment operators*.

Edges removed in:	Adol-C	COS	matrix	vkey	Edraw	Graphdr	Drawserv
Briand-A	73	16	88	85	84	539	3497
Briand-B	77	17	88	92	97	539	3497
COS-A (9999,2,9999,2,2,9999)	86	16	88	75	80	504	3540
COS-A (2,2,2,2,2,2)	39	15	88	50	56	237	1489

small in all test cases. We conclude that using Briand-A is preferable due to its single pass, and in the remainder of the results described here we utilize Briand-A as the representative of the Briand approach.

Comparing the lines in Table 6 corresponding to Briand-A and to COS-A with costs (9999, 2, 9999, 2, 2, 9999) allows a direct comparison of the two methods when they are applied only to ADP edges. Those results show no substantial difference between the two methods, with Briand-A being superior in two test cases, COS-A being superior in three test cases and the two methods being equal in one test case. Since both methods take the same general approach in breaking the ORD into strongly connected components and then selecting edges to break the cycles in those SCCs, the equality may not be a surprise. On the other hand, the two methods select the edges for removal in very different ways. Recall that the Briand weights are based on the product of in and out degrees, while the COS weights are based on the edge types.

The third aspect of comparison arising from Table 6 comes in comparing Briand-A to COS-A with costs (2,2,2,2,2,2). This measures the advantage that can be gained in terms of the total number of stubs that need to be written by allowing the stubbing of all types of edges, rather than just ADP edges. And, the reduction in the number of stubs is quite significant in five of the seven test cases, ranging from 33% to 64% reductions. Further, the largest percentage decrease also occurred in the test case having the largest absolute number of stubs, dropping from 3497 stubs for Briand-A down to 1489 stubs for COS-A. The only test cases where there was not a significant difference between Briand-A and COS-A were for COS, where the number of edges to be removed is small (16 for Briand-A and 15 for COS-A), and for matrix where the ORD contains *no* inheritance edges, hence both algorithms remove only ADP edges.

Table 4. *Edge types removed by COS-A for cost model (20,2,4,2,2,4).*

Number of:	Adol-C	COS	matrix	vkey	Edraw	Graphdr	Drawserv
Inheritance edges	0	0	0	0	0	1	15
Association edges	0	1	0	19	1	12	36
Composition edges	0	0	0	0	0	0	0
Dependency edges	30	13	88	22	28	75	381
Polymorphic edges	56	2	0	34	51	406	2356
Owned Element edges	0	0	0	0	0	0	0

Table 5. *Number of removed edges for several cost models.* The entry for (80,2,4,2,2,4) is identical to that for (40,2,4,2,2,4) and is omitted. The numbers in parentheses indicate the number of inheritance edges that were removed.

Edges removed in:	Adol-C	COS	matrix	vkey	Edraw	Graphdraw	Drawserv
Briand-A	73	16	88	85	84	539	3497
COS-A (40,2,4,2,2,4)	86 (0)	16 (0)	88 (0)	75 (0)	80 (0)	504 (0)	2900 (9)
COS-A (20,2,4,2,2,4)	86 (0)	16 (0)	88 (0)	75 (0)	80 (0)	494 (1)	2788 (15)
COS-A (10,2,4,2,2,4)	68 (2)	16 (0)	88 (0)	75 (0)	74 (1)	430 (4)	2653 (20)
COS-A (5,2,4,2,2,4)	46 (6)	16 (0)	88 (0)	68 (1)	56 (3)	307 (10)	2184 (51)
COS-A (2,2,20,5,20,20)	37 (7)	15 (2)	88 (0)	49 (10)	72 (8)	183 (26)	1073 (82)

We also note that by using different COS cost models, it is possible to create even fewer total stubs. For instance, using COS-A with the cost model (2,2,20,5,20,20), only 1073 edges are removed for Drawserv (see the last row, last column of Table 6.1). The argument in favor of breaking ADP edges has of course been that inheritance edges are difficult to stub. From these results it seems that rethinking the stubbing of inheritance edges is in order.

6.2 Minimizing the overall cost of stubbing

Tables 6.1 and 6.1

provide insights into the overall cost of stubbing the programs in our test suite.

Table 6.1 shows, for one cost model, the number of edges of each type that are removed by COS-A. There, it is interesting to note the dominance of ADP edges in all of the test cases. This is not surprising since that cost model assigns a weight of 2 to ADP edges, and a weight of 20 to inheritance edges. Even so, for the largest application Drawserv, COS-A selected 15 inheritance edges for removal, and this resulted in removing about 700 fewer ADP edges than if no inheritance edges were removed (from Table 6). Note also that no composition edges were selected for removal in any test case, since as noted earlier, no composition edges are located in a cycle in any test case.

Table 6.1 shows the total number of edges and (in parentheses) the number of inheritance edges removed for a set of cost models that vary the cost of inheritance edges. From this figure there are three observations. The first is that as the cost of the inheritance edges decreases, the number of inheritance edges removed increases, and the total number of stubs decreases, as many fewer ADP edges are removed. The second observation, is that by removing just a few inheritance edges, the number of stubbed ADP edges (and the total stubbed edges) may drop dramatically. For the Drawserv test case, Briand-A removes 3497 edges, while COS-A with a cost of 40 for inheritance edges, removes only 9 inheritance edges, and yet the total number of removed edges drops to 2908. That is a savings of almost 600 edges! Even if inheritance edges are difficult to stub, it seems that creating 9 stubs versus 600 stubs is a potentially large advantage. The third observation is that by using different cost models the overall number of removed edges can drop dramatically, as in the final line of that table; the cost model for this final line of results also appeared in reference (Malloy et al., 2003a). There, the total number of removed edges is just under 1100, and there are 82 removed inheritance edges. The trade-off between the edges removed here, and those removed when only ADP edges are considered is dramatic: stubbing 991 ADP edges and 82 inheritance edges versus stubbing about 3500 ADP edges. In terms of actually creating stubs, the savings of the former over the latter is considerable, even allowing for the possibility that inheritance edges may be an order of magnitude more difficult to stub.

6.3 Which version of COS?

The results in Table 6 allow a comparison of the effectiveness of the three versions of the COS algorithm. In this table we show the number of edges (individual edges) removed by the COS variants using two cost models. As is apparent from those numbers there are only slight differences between the three versions. For any given test case and cost model, any one of the three variants might break fewer edges than the other two, and in most cases the differences are small (a few percent or so). Below we consider why there are generally no appreciable differences in the performance of the three versions.

Table 6. *Comparing the variants of COS - Removed edges.*

Number of:	Adol-C	COS	matrix	vkey	Edraw	Graphdraw	Drawserv
Cost Model (2,2,2,2,2,2)							
COS-A (arbitrary)	39	15	88	50	56	237	1489
COS-R (random)	39	15	88	48	76	236	1612
COS-B (Briand)	39	15	88	46	56	216	1262
Cost Model (40,2,4,2,2,4)							
COS-A (arbitrary)	86	16	88	75	80	504	2900
COS-R (random)	86	17	88	76	103	538	2960
COS-B (Briand)	86	16	88	70	80	514	2908

First consider COS-R (which selects a random minimum weight edge) in comparison with COS-A (which selects a minimum weight edge incident to the lowest index node). Although COS-A favors the selection of edges incident to nodes of lower index, while COS-R tends to spread the selection among all edges, the bottom line is that ultimately all cycles need to be broken. The edges removed to accomplish this are not particularly relevant - if an edge is in a cycle, then it can be removed. Spreading the edges across the ORD seems not to be an advantage, and in fact, selecting multiple edges associated with a single node may be an advantage to the programmer when they are writing the actual stubs (i.e. it is probably easier to stub six methods associated with one class rather than one method in each of six classes, since the former requires understanding one class versus understanding six in the latter case). Thus, COS-A is preferred to COS-R due to ease of implementation.

The same conclusion can be reached in comparing COS-A with COS-B (which utilizes a secondary weight to select among minimum weight edges). Although it would seem to be advantageous to remove edges based on a measure of the number of cycles containing that edge, it seems not to help in this case. This apparently stems from the fact that in the COS ORD with merged edges, the majority of the nodes have low incoming and outgoing degrees. Specifically, taken over all seven test cases, our data (not shown here) shows that 65% of the nodes have indegrees of 3 or less, and 69% of the nodes have outdegrees of 3 or less. As a result, the secondary weights of many edges tend to be similar in value. This means that there is not much distinction between the edges in terms of their ability to break cycles. Since the secondary weight is only an estimate of the ability of the edge to break cycles, small differences appear

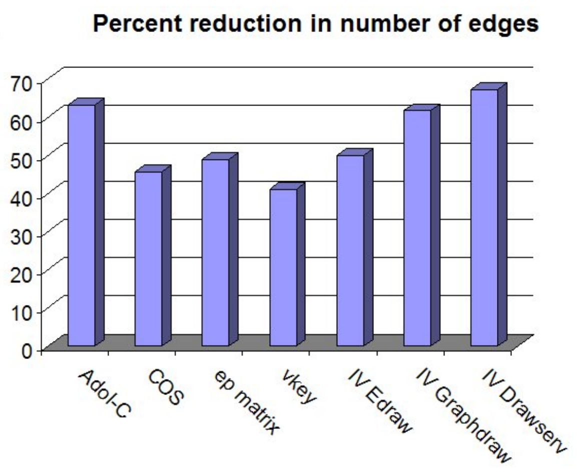


Fig. 7. *Reduction in the size of the COS ORD.* This figure illustrates the percentage reduction in the total number of edges in a COS ORD when edges of the same type with the same source and destination are merged.

not to be significant.

6.4 What about running times?

One unanticipated advantage of the COS system as compared with the Briand system, is that the running time of the COS algorithms is considerably faster than that of the Briand algorithm. This advantage arises because the COS algorithms work with the COS ORD and its *merged* edges, rather than with the full set of individual edges appearing in a standard ORD. This advantage manifests itself in two ways.

First, the size of the COS ORD manipulated by COS is smaller than a standard ORD. Figure 7 illustrates the reduction in the the number of edges in a COS ORD as compared to a standard ORD. In the figure, the test cases are listed along the X axis, and the percentage reduction in the number of edges in the COS ORD is listed along the Y axis. For example, for the Adol-C test case there were 111 individual edges and 41 merged edges; the first bar of the graph in Figure 7 shows a 63% reduction in the size of the COS ORD as compared to a standard ORD. For the Drawserv test case, there were 6460 individual edges and 2118 merged edges; the last bar of the graph in Figure 7 shows a 67% reduction in the size of the COS ORD as compared to a standard ORD. The reduction in the number of edges in the COS ORD as compared to a standard ORD averages 54%. This reduction in the number of edges is most advantagous for the last two test cases shown in the graph, since they have the most edges, Graphdraw and Drawserv. There, the size of the COS ORDs were reduced by 62% and 67% respectively over the number of edges in standard

ORDs. As a result of using fewer edges, when computing strongly connected components, the time required is proportionally less for the COS ORD.

The second advantage of using merged edges is that the COS approach determines which stubs are necessary based on the removal of merged edges. That is, when a merged edge e is removed from the ORD, the methods associated with the individual edges that were merged to create edge e are stubbed. Thus, a number of stubs are associated with a single merged edge. The advantage is that COS removes only that one merged edge, whereas the Briand methods would remove the individual edges one at a time. Recall that in both Briand and COS, the removal of an edge causes the recomputation of the SCCs where the edge was removed. Each such SCC computation requires time proportional to the number of nodes and edges in that SCC. By removing merged edges the number of such recomputations is significantly smaller in COS than when individual edges are removed in Briand.

Table 6.4 shows a comparison of the number of edges removed by the two algorithms (individual edges by Briand-A, merged edges by COS-A) using two cost models. For the larger test cases, the differences in the number of removed edges is significant, running as high as 4:1. Since the running times of both algorithms are proportional to the number of removed edges, the COS algorithm has a significant running time advantage. The advantage is magnified by the fact that the COS ORD is also significantly smaller than a standard ORD. Timing results shown in Table 6.4 on actual runs of the two algorithms confirm this advantage. In that figure, timings are shown for Briand-A and for three representative cost models in conjunction with COS-A. The measurement for these timings begins immediately after the common step of constructing the ORD.

Although the specific time advantage of COS-A over Briand-A varies with the particular cost model, in all cases the running time of COS-A was significantly lower than the running time of Briand-A. For example, for the first of three COS-A models listed in Table 6.4, consider the model with parameters (9999,2,9999,2,2,9999) for the Drawserv test case: the running time for this model is 292,870 *ms* as compared to 862,590 *ms* for the Briand-A model, which means that the running time for the COS-A model is only 33% of the running time for the Briand-A model. Moreover, for the COS-A model with parameters (2,2,20,5,20,20) for the Drawserv test case: the running time for this model is 37,790 *ms* as compared to the 862,590 *ms* for the Briand-A model, which means that the running time for the COS-A model is only 4% of the running time for the Briand-A model.

Table 7. *Number of removed edges: individual versus merged.*

Number of:	Adol-C	COS	matrix	vkey	Edraw	Graphdr	Drawserv
Individual edges (Briand-A)	73	16	88	85	84	539	3497
Cost Model (2,2,2,2,2,2)							
Merged edges (COS-A)	20	9	34	37	29	128	873
Cost Model (80,2,4,2,2,4)							
Merged edges (COS-A)	27	9	34	53	32	194	1321

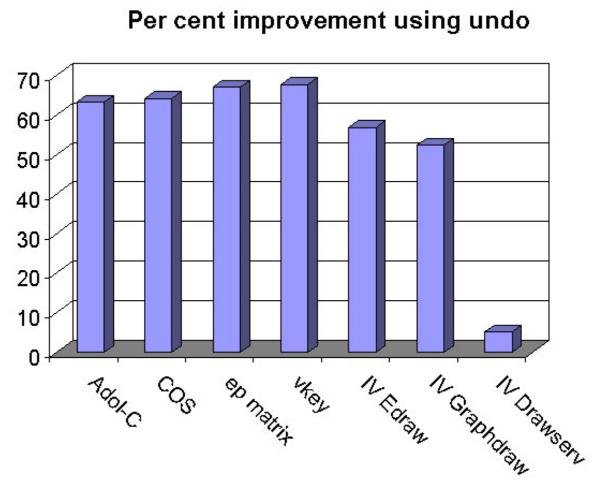
Table 8. *Running times.*

Running time (ms):	Adol-C	COS	matrix	vkey	Edraw	Graphdraw	Drawserv
Briand-A	90	20	30	340	200	8600	862590
COS-A (9999,2,9999,2,2,9999)	30	10	20	180	60	2830	292870
COS-A (2,2,2,2,2,2)	20	10	20	80	60	1790	172390
COS-A (2,2,20,5,20,20)	10	10	10	70	100	840	37790

6.5 The Command pattern: the performance effects of undo

Figure 8 shows a comparison of the total running times for **COS-A** using a series of three cost models with and without the undo functionality of the Command pattern in the COS. The timings in this figure measure execution time for all three stages of the ordering methodologies, including the time to construct the ORD, break all cycles and compute a class ordering. In constrast, the timings shown in Table 6.4 do not include the time to build an ORD since that time is the same for all methodologies. The first row of the table in Figure 8 lists each of the seven test cases. The second and third rows of the table list timings for three iterations through the COS-A method without undo and with undo, respectively. The final row of the table lists timings for the second stage of the methodology: the time spent in SCC computation.

The second and third rows of the table in Figure 8 illustrate the time savings advantage of the undo functionality in the Command pattern. For example, consider the running time of the Adol-C test case without undo, 1448 ms (all timings are in milliseconds), and with undo, 532 ms, as illustrated in



Total run time (ms):	Adol-C	COS	matrix	vkey	Edraw	Graphdraw	Drawserv
Without *undo*	1448	770	125450	60945	1684	26645	551040
With *undo*	532	276	41268	19624	710	12692	523210
SCC Computation	44	24	26	218	148	4138	355470

Fig. 8. *Total running times of COS-A for cost models (9999,2,9999,2,2,9999), (2,2,2,2,2,2), and (2,2,20,5,20,20):* with and without undo

the second and third rows. The execution time using undo is almost three times faster than the time without using undo, or a 63% improvement in execution time. The bar chart at the top of Figure 8 further emphasizes this improvement. There are similar improvements for the other test cases with the best improvement of 68% for the vkey test case. The exception to this improvement in execution time is the Drawserv test case where the execution time without undo is 551,040 ms compared to 523,210 ms with undo: using undo gained only 5% improvement over the execution time without undo. This small improvement derives from the fact that the SCCs for this test case are very large and the time to build the SCCs dominated the computation time. The time to compute and recompute the SCCs for the Drawserv test case across all three cost models is 355,470 ms, which dominated the total running time and eroded the time savings from using the undo to eliminate the time to re-build the ORD during the cycle breaking process.

7 Concluding Remarks

We have presented the design and implementation of a system that exploits well-known design patterns to facilitate construction of an extensible system for comparison and visualization of ordering methodologies for class-based testing of C++ applications. Using the design patterns together with the *dot* tool from the Graphviz package (AT&T Labs, 2005), we incorporated visualization of the ORD and the edge removals into our system. This visualization enabled detailed presentation and graphical visualization, and facilitated comprehension, debugging and validation of the ORD and edge removal process.

There are four ordering methodologies presented in the literature; three of the methodologies are based on removal of edges to break cycles (Briand et al., 2001; Kung et al., 1995; Labiche et al., 2000) and one methodology is based on removal of nodes (Tai and Daniels, 1997). Of the three methodologies based on edge removal, the methodology of Briand et al. (Briand et al., 2001) subsumes that of Kung et al. (Kung et al., 1995), and Briand et al. (Briand et al., 2001) demonstrated that their methodology is better than that of Labiche et al. (Labiche et al., 2000). Thus, using our implemented system, we have also presented a comparative study and evaluation of the remaining methodology of Briand, et al. (Briand et al., 2001) and a new methodology, COS, for generating an integration order of classes to minimize the cost of stub construction. Our study addresses the identified need for empirical evaluation and comparison of testing strategies (Do et al., 2004; Harrold, 2000; NIST, 2002; Orso et al., 2004). We have also demonstrated the extensibility of our COS system to both visualization and node-based removal methodologies.

All methodologies, except for Labiche et al. (Labiche et al., 2000) and COS, fail to include polymorphic edges in their ORD and cycle breaking considerations. By not including polymorphic edges, dependencies between classes are omitted and the generated class ordering is incorrect. Thus, to generate a correct class order and to ensure that the cycle breaking algorithm terminates, the approach of Briand, et al. (Briand et al., 2001) required modification to include polymorphic edges in the ORD and edge removal considerations.

The main results of the comparative study of Briand, et al. (Briand et al., 2001) and COS were:

- The number of stubs can be drastically reduced by allowing the stubbing of inheritance edges. Using the COS cost model approach, the trade-off between the number of removed inheritance edges and the number of removed ADP edges can be controlled. Based on these results, additional research into the stubbing of inheritance edges seems appropriate.
- When restricted to the removal of ADP edges, the Briand approach and

COS were equivalent in the number of removed edges.
- The running time of the COS algorithms is considerably less than those of Briand, due primarily to the use of *merged* edges. Similarly, the size of the COS ORD is considerably smaller than an ordinary ORD.
- The three variants of COS perform equivalently in terms of the number of edges removed, thus COS-A is preferred due to having the simplest implementation.

The code for the COS implementation is available upon request or from the first author's web page.

References

A. Griewank and O. Vogel, April 2003. http://www.math.tu-dresden.de/wir/project/adolc/.

Aho, A. V., Hopcroft, J. E., Ullman, J. D., 1974. The Design and Analysis of Computer Algorithms, 2nd Edition. Addison-Wesley.

AT&T Labs, June 2005. Graphviz. http://www.research.att.com/sw/tools/graphviz/.

Binder, R., 2000. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley.

Briand, L., Feng, J., Labiche, Y., July 2002. Using genetic algorithms and coupling measures to devise optimal integration test orders. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering. IEEE Computer Society Press, Ischia, Italy, pp. 43–50.

Briand, L., Labiche, Y., Wang, Y., Nov. 2001. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In: Proceedings of the 12th International Symposium on Reliability Engineering (ISSRE '01). IEEE Computer Society Press, pp. 287–297.

Daniels, F. J., Tai, K. C., 1999. Measuring the effectiveness of method test sequences derived from sequencing constraints. TBD, 74–83.

Do, H., Rothermel, G., Elbaum, S., Jan 2004. Infrastructure support for controlled experimentation with software testing and regression testing techniques. Technical Report 04-60-01.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Garey, M. R., Johnson, D. S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness, 1st Edition. Freeman and Company.

Harrold, M. J., June 2000. Testing: A roadmap. In: Proceedings of the International Conference on Software Engineering, ICSE200. Limerick, Ireland.

ISO/IEC JTC 1, September 1998. International Standard: Programming Languages - C++, 1st Edition. No. 14882:1998(E) in ASC X3. ANSI.

Karp, R. M., 1979. Reducibility among combinatorial problems. In: Miller, R. E., Thatcher, J. W. (Eds.), Complexity of Computer Computations. Plenum Press.

Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., august 1995. A test strategy for object-oriented programs. In: Proceedings of the 19th International Computer Software and Applications Conference (COMPSAC'95). IEEE Computer Society Press, pp. 239–244.

Labiche, Y., Thevenod-Fosse, P., Waeselynck, H., Durand, M. H., June 2000. Testing levels for object-oriented software. In: ICSE. New York, pp. 136–145.

Le Traon, Y., Jeron, T., Jezequel, J.-M., Morel, P., March 2000. Efficient object-oriented integration and regression testing. IEEE Transactions on Reliability 49 (1), 12–25.

Lloyd, E. L., Malloy, B. A., 2005. A study of test coverage adequacy in the presence of stubs. Journal of Object Technology 4 (5), 117–137.

Lu, C., Chu, W., 2003. Integrating diverse paradigms in evolution and maintenance by an xml-based unified model. Journal of Software Maintenance and Evolution 15, 111–144.

Malloy, B. A., Clarke, P. J., Lloyd, E. L., Nov. 2003a. A parameterized cost model to order classes for class-based testing of c++ applications. In: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03). IEEE Computer Society Press, pp. 353–364.

Malloy, B. A., Gibbs, T. H., Power, J. F., 2003b. Decorating tokens to facilitate recognition of ambiguous language constructs. Software, Practice & Experience 33 (1), 19–39.

Martin, R. C., 2003. Agile Software Development. Prentice Hall, 0-13-597444-5.

Matzko, S., Clarke, P., Gibbs, T. H., Malloy, B. A., Power, J. F., Feb 2002. Reveal: A tool to reverse engineer class diagrams. In: Proceedings of the International Conference on the Technology of Object-Oriented Languages and Systems. Sydney, Australia.

NIST, May 2002. The economic impacts of inadequate infrastructure for software testing. Technical Report.

OMG Unified Modeling Language Specification, March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.pdf.

Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., Harrold, M. J., May 2004. An empirical comparison of dynamic impact analysis algorithms. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004). pp. 491–500.

Shoup, V., March 2002. Number theory library. http://www.shoup.net/ntl/.

Stroustrup, B., 1997. The C++ Programming Language, 3rd Edition. Addison-Wesley.

Tai, K.-C., Daniels, F. J., 1997. Test order for inter-class integration testing of object-oriented software. In: 21st International Computer Software and Applications Conference, COMPSAC'97. IEEE, pp. 602–607.

Tarjan, R., 1972. Depth-first search and linear graph algorithms. SIAM Journal on Computing 1 (2), 146–160.

Vlissides, J. M., Linton, M. A., March 2002. IV tools. http://www.vectaport.com/ivtools/.

Wampler, B., October 2001. The V C++ GUI framework. http://www.objectcentral.com.

Appendix A

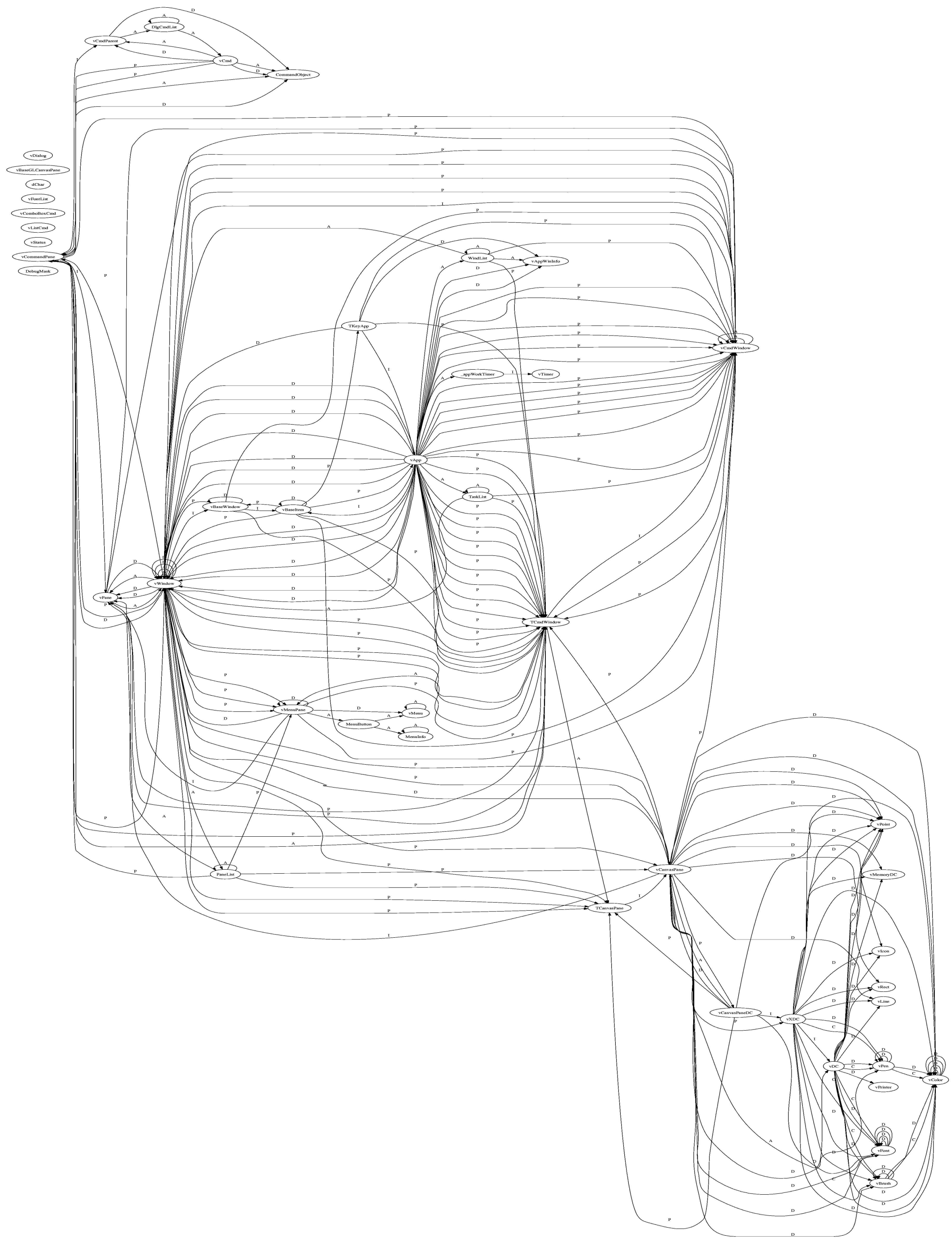


Fig. 9. *Visualizing cycles*. The ORD in this figure is generated automatically by our GraphvizVisitor and the *dot* tool, providing a visualization overview of the cycles in the ORD for the vkey test case.