

# Coupling-based Class Integration and Test Order\*

Aynur Abdurazik  
Information and Software Engineering  
George Mason University  
Fairfax, VA 22030, USA  
aabduraz@gmu.edu

Jeff Offutt  
Information and Software Engineering  
George Mason University  
Fairfax, VA 22030, USA  
offutt@ise.gmu.edu

## ABSTRACT

During component-based and object-oriented software development, software classes exhibit relationships that complicate integration, including method calls, inheritance, and aggregation. When classes are integrated and tested, an *order* of integration must be established. The difficulty arises when cyclic dependencies exist – the functionality that is used by the first class to be tested must be mimicked by creating “stubs” (sometimes called “mocks”), an expensive and error-prone operation. This problem is generally called the *class integration and test order* (CITO) problem, and solutions must be fully automated for integration and testing to proceed smoothly and efficiently. This paper describes new techniques and algorithms to solve the CITO problem. New results include improved edge *weights* that are derived from quantitative coupling measures to more precisely model the cost of stubbing, and the use of weights on nodes, allowing more information to be used. Also, a new algorithm for computing the integration and test orders is presented. The technique is compared with an existing approach with positive results.

## Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools*; D.2.8 [Software Engineering]: Metrics—*Complexity Metrics*

## General Terms

Measurement, Verification

## Keywords

OO testing, class integration and test order, coupling

\*The authors are sponsored in part by National Institute of Standards and Technology (NIST), Software Diagnostics and Conformance Testing Division (SDCT). This work was sponsored in part by the National Science Foundation under grant number CCR-0097056.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AST'06, May 23, 2006, Shanghai, China.

Copyright 2006 ACM 1-59593-085-X/06/0005 ...\$5.00.

## 1. INTRODUCTION

A common problem in inter-class integration testing of object-oriented software is to determine the order in which classes are integrated and tested [6]. When one class requires another to be available before it can be executed, we define this kind of relationship to be a *dependency*. These two classes can be characterized as *server* and *client* classes. The client class is being compiled or executed and the server class must be present. This dependency has a direction, and is based on one or more OO relationships.

When there is no cycle in the dependency of classes or subsystems, the *class integration and test order* (CITO) problem can be solved by a simple reverse topological ordering of classes based on their dependencies. However, dependency cycles are common in real-world systems and when present, topological sorting is not possible [6].

To solve the CITO problem in the presence of cycles, the cycles must be broken. The effect of breaking a dependency cycle is that a *stub* must be created for the class that is no longer present, thus increasing the cost of integration testing. Our goal is to find an optimal order that minimizes the stubbing effort. Stubbing effort has many elements to consider, and, therefore, cannot be completely measured or estimated [2]. It has also been suggested that creating stubs can be error prone and costly [1].

Coupling measurement captures class relationships. The main goal of this study is to use coupling measurement to estimate stubbing effort and develop an efficient technique for finding an *optimal integration order*.

The remainder of the paper discusses existing solutions, introduces our model, then presents results from a case study taken from Briand et al.'s paper [2].

## 2. SUMMARY OF EXISTING SOLUTIONS

The class integration and test order problem has been addressed by several researchers and several solutions have been proposed. The solutions can be categorized into *graph-based* and *genetic algorithm-based* approaches. This section summarizes existing solutions and discusses their advantages and disadvantages.

In graph-based approaches, classes and their relationships in software are modeled as object relation diagrams (ORD) or test dependency graphs (TDG). An ORD or TDG is a directed graph  $G(V, E)$  where  $V$  is a set of nodes representing classes and  $E$  is a set of edges representing the relationships among classes. The class integration and test order problem is to find an ordering of nodes in the graph so that the classes can be integrated and tested with minimum effort.

In most papers [4, 6, 8, 9], the testing effort is estimated by counting the number of test stubs that need to be created during integration testing. This method assumes that all stubs are equally difficult to write. One recent paper tries to consider test stub complexity when estimating the testing effort [7].

In the genetic algorithm-based approach [2], inter-class coupling measurements and genetic algorithms are used in combination to assess the complexity of test stubs and to minimize complex cost functions.

Kung et al. [6] were the first researchers to address the class test order problem and they showed that, when no dependency cycles are present among classes, deriving an integration order is equivalent to performing a topological sorting of classes based on their dependency graph – a well known graph theory problem. In the presence of dependency cycles, they proposed a strategy of identifying strongly connected components (SCCs) and removing associations until no cycles remain. When there is more than one candidate for cycle breaking, Kung et al.’s approach chooses randomly. They mention that a possible solution would involve the use of the complexity of the associations involved in cycles.

Tai and Daniels proposed a number of properties for inter-class test ordering [8]. They assumed that aggregation and inheritance relations do not form cycles, but association relations may. Tai and Daniels defined a *major* and a *minor* level for classes, and sorted classes according to these levels. First, classes are assigned major level numbers according to the inheritance and aggregation relationships only. Then, within each major level, minor-level numbers are assigned based on the association relationships only. In this case, first, strongly connected components (SCCs) in a major level are identified, then each edge in a SCC is assigned a value, called *weight* ( $e$ ), which is defined as the sum of the number of incoming dependencies of the origin node of  $e$  and the number of outgoing dependencies of the target node of  $e$ . Edges with higher values are selected to break cycles. The hypothesis is that removing edges with higher values will break more cycles. However, Briand et al. [4] showed this hypothesis is not always true. Another problem is that their algorithm may break an association edge that crosses major levels but is not involved in any cycles [4].

Le Traon et al. assigned weights to each node in the ORD, then removed the incoming edges of the node with maximum weight [9]. This process is repeated until no cycle remains in the ORD. To assign weights, they first used Tarjan’s algorithm to identify strongly connected components. In each SCC, edges are partitioned into *tree edges*, *forward edges*, *frond edges*, and *cross edges*. The weight of a node is the sum of the number of incoming and outgoing frond edges.

Le Traon et al.’s approach is non-deterministic in two ways. First, different sets of edges can be labeled as *frond edges* depending on the different starting node. Second, the approach arbitrarily chooses a node when two or more nodes have the same weight. Thus, different runs of the algorithm result in different outcomes.

Briand et al. [3, 4] proposed a graph-based strategy for ordering classes for testing that combines Tai and Daniels and Le Traon et al.’s approaches. They first used Tarjan’s algorithm to identify strongly connected components (SCCs). Next, weights are assigned to *association* edges in the SCCs. The weight of an edge is the *estimated* number of cycles that the edge may be involved in. Let  $G_i(V_i, E_i)$  be a SCC of

graph  $G(V, E)$  and  $v_1, v_2 \in V_i, e \in E_i$ , and  $e = v_1 \rightarrow v_2$ . The estimated weight of edge  $e$  is  $weight(e) = (v_1)_{in} \times (v_2)_{out}$ , where  $(v_1)_{in}$  is the number of incoming dependencies of node  $v_1$  and  $(v_2)_{out}$  is the number of outgoing dependencies of node  $v_2$ . Then, the edge with the highest weight value is removed. These steps are repeated until no SCC remains.

Briand et al.’s approach has the advantage over Le Traon’s approach of not breaking inheritance and aggregation edges and also the weight computation for edges is more precise than Tai and Daniels’ approach.

Subsequently, Briand et al. [2] use a genetic algorithm and coupling metric to try to break cycles by removing edges that will reduce the complexity of stub construction. A *genetic algorithm* is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. It is a search algorithm that locates optimal binary strings by processing an initially random population of strings using artificial mutation, crossover and selection operators, in an analogy with the process of natural selection. Briand et al. conclude that composition and inheritance relationships should never be removed since, according to their heuristic, removal of these edges would likely to lead to complex stubs. The complexity of stub construction for parent classes is induced by the likely construction of stubs for most of the inherited member functions [3]; moreover, inherited member functions must be tested in the new context of the derived class rather than the context of the parent class [5]. Their experiment showed that genetic algorithms can be used to obtain optimal results by using more complex cost functions and perform as well as graph-based algorithms under similar conditions.

Malloy et al. developed a *Class Ordering System* that is driven by a parameterized cost model [7]. They used a strategy similar to Briand et al.’s graph-based approach [4]. They defined six types of edges, *association*, *composition*, *dependency*, *inheritance*, *owned element*, and *polymorphic*. These edges are assigned weights of (2, 2, 20, 5, 20, 20) based on their estimation of the cost of stub construction for untested classes based on heuristics. For an ORD  $G = (V, E)$ , where  $V$  is a set of nodes representing classes and  $E$  is a set edges representing relationships among classes, their cost model  $C = \langle W, f(e), w(m_{x,y}) \rangle$  is a 3-tuple where  $W$  is a set of weight assignments and  $f(e)$  and  $w(m_{x,y})$  are weight functions. When there is a cycle, the edge with the smallest weight is removed from the strongly connected component. When there is no cycle, the reverse topological sort of the nodes in the ORD is the order for integration test.

To summarize, the existing graph-based approaches use high level, course grained, estimates of test stub complexity. The GA approach must be run many times, greatly complicating the process.

### 3. A NEW MODEL AND ALGORITHMS

This section introduces a new graph-based solution for the CITO problem. Our approach is different from other graph-based approaches in three respects. First, we model classes and their relationships with weights on both nodes and edges. Second, weights of nodes and edges are based on a quantitative measure of coupling. Last, we use an algorithm that incorporates edge and node weights as well as number of cycles in cycle breaking. For space reasons, this paper presents the algorithm using only edge weights.

### 3.1 Modeling Class Integration & Test Order

As said in section 2, dependencies among classes are usually modeled in graphs. The CITO problem then becomes finding an acyclic graph with minimum cost. The cost is usually modeled as the number of test stubs to be generated during the testing activities. This section uses a different abstraction for test dependencies among classes and a different cost model for the testing effort. We model the test dependencies among classes using a *Weighted Object Relation Diagram (WORD)*, and model the testing effort by computing test stub complexities using coupling information.

To develop more precise testing and maintenance predictive models, we defined nine types of couplings based on explicit object-oriented class relationships and some implicit relationships. They are *Association Coupling*, *Aggregation Coupling*, *Composition Coupling*, *Usage Dependency Call Coupling*, *Global Coupling*, *Inheritance Coupling*, *Interface Realization Coupling*, *External Coupling*, and *Exception Coupling*. For each coupling type, coupling measures are defined to measure the dependencies between a server and client class in terms of the number of distinct variables used, the number of distinct methods (including constructors) called, the number of parameters sent, and the number of return value types. The coupling measures also include a coupling type indicator. The following equation uses a “dot” notation to represent a *coupling measure (CM)* for couplings between two classes  $c_i$  and  $c_j$ :

$$CM(c_i, c_j) = C.V_d.M_d.R_d.P_d, \quad (1)$$

where  $c_i$  and  $c_j$  represent two classes that are coupled together,

$$C = \begin{cases} 5 & \text{when coupling is based on inheritance} \\ & \text{or composition} \\ 1 & \text{for other coupling types} \end{cases}$$

The dot notation is used to indicate that the five measures are independent but related.  $V_d$ , *number of distinct vars*, represents the number of distinct public vars of  $c_j$  that are directly used by  $c_i$ .  $M_d$ , *number of distinct methods*, represents the number of distinct public methods of  $c_j$  that are called by  $c_i$ .  $R_d$ , *number of distinct return types*, represents the number of distinct return types that appear in  $M_d$ .  $P_{dist}$ , *number of distinct parameters*, represents the number of distinct parameters that appear in  $M_d$ .

Equation 1 assigns different values to  $C$  to distinguish different coupling types in our measure so that they can be analyzed and used methodically when needed. We observe that these coupling measures can syntactically estimate the content of a test stub needed by a client class. Thus, using the coupling measures, a *test stub complexity* for each class can be estimated in the context of whole system.

We define two kinds of stub complexity. If a client class A depends on a server class B to function, we can quantify this dependency by identifying the scope of B used by A, as measured by coupling. We define this to be a *specific test stub complexity* of B to A. However, there can be other client classes that depend on B, and some usage of B can be overlapped among client classes. We define the sum of dependencies/usages from all other client classes to B as the *total stub complexity* of server class B. When we compute the total stub complexity of a class, we will take into account the overlapping possibility of specific stubs. Thus, a total stub

complexity of a class takes a value between the maximum and sum of several specific stubs complexity.

We use the *specific test stub complexity* and *total stub complexity* to assign weights to edges and nodes in our WORD. In a WORD, nodes represent classes and edges represent test dependencies among classes. Both nodes and edges are weighted. The node weight represents the *total stub complexity* of a class, and the edge weight represents the *specific test stub complexity* of a server class to the client class that is connected by the edge. The graph can be acyclic or cyclic. If the graph is acyclic, then we can carry out integration testing in the reverse topological order of the graph. If the graph is cyclic, then we have to break cycles first. This causes the introduction of test stubs for the broken edges or removed nodes. We model the testing effort as the total complexity of stubs that are introduced during integration testing. The goal is to make the graph acyclic by removing certain edges and/or nodes, and the total weight of removed edges and/or nodes has to be minimum.

The model for the class integration and test order problem is formally defined as follows:

Let  $G(V, E)$  be a node- and edge-weighted directed graph that models classes or components and their relationships. In the graph, nodes represent classes or components, and edges represent test dependencies among classes. The edge weights represent *specific test complexities* and node weights represent *total stub complexities*. Our problem is to determine the nodes and edges with minimum total weight to remove so that there are no cycles in  $G$ .

#### 3.1.1 Measuring Stub Complexity

We use coupling measures to assign weights to edges and nodes in our weighted object relation diagram (WORD). After all couplings are measured in the form of equation 1, coupling measures between the source and target node classes are aggregated as one measure as follows:

$$\begin{aligned} cm_{e_i} &= \left\{ \sum_{k=1}^9 CM_k(v_m, v_n) \right. \\ &\quad \left. | v_m, v_n \in V, e_i = v_m \rightarrow v_n, e_i \in E \right\} \\ &= \max(C) \cdot \sum_{k=1}^9 V_{d_k} \cdot \sum_{k=1}^9 M_{d_k} \cdot \sum_{k=1}^9 R_{d_k} \cdot \sum_{k=1}^9 P_{d_k} \quad (2) \end{aligned}$$

This measure will be used to compute the weight of an edge and represents a *specific test stub complexity* of a class.

The coupling measures on edges are then further aggregated to nodes. A coupling measure on a node is computed from the coupling measures of the incoming edges of the node in the following manner:

$$\begin{aligned} cm_{v_i} &= \left\{ \left[ \max(cm_{e_{1,i}}, cm_{e_{2,i}}, \dots, cm_{e_{k,i}}), \sum_{l=1}^k cm_{e_{l,i}} \right] \right. \\ &\quad \left. | v_i \in V, e_{l,i} = v_l \rightarrow v_i, e_{l,i} \in E, |e_{l,i}| = k \right\} \quad (3) \end{aligned}$$

where  $cm_{e_{1,i}}, cm_{e_{2,i}}, \dots, cm_{e_{k,i}}$  are coupling measures on the incoming edges of node  $v_i$  and the summation of coupling measures are the same as in equation 2. Equation  $cm_{v_i}$  takes a value between the maximum and sum of coupling measures on the incoming edges of node  $v_i$ . This measure will be used to compute the weight of a node and represents the *total test stub complexity* of a class.

Our rationale for introducing weights for nodes is that

specific stubs for a class may overlap. It is possible that certain methods or variables of a server class can be used by a number of clients in the same way. In this case, creating one stub for the server class can satisfy the needs of several clients.

We use Briand et al.'s method for estimating stubbing complexity [2] from the coupling measures of edges and nodes. For a measure  $Cplx()$ , a complexity measure  $\overline{Cplx}()$  is normalized as

$$\overline{Cplx}(i, j) = Cplx(i, j) / (Cplx_{max} - Cplx_{min}) \quad (4)$$

where  $Cplx(i, j)$  represents a complexity information matrix,  $Cplx_{min} = \text{Min}\{Cplx(i, j), i, j = 1, 2, \dots\}$  and  $Cplx_{max} = \text{Max}\{Cplx(i, j), i, j = 1, 2, \dots\}$ . They use two coupling measures  $A()$  and  $M()$ , the number of locally defined variables and the number of methods, to compute overall stubbing complexity:

$$SCplx(i, j) = (W_A \cdot \overline{A}(i, j)^2 + W_M \cdot \overline{M}(i, j)^2)^{1/2} \quad (5)$$

where  $W_A$  and  $W_M$  are weights and  $W_A + W_M = 1$ . Thus, for a given test order  $o$ , with  $d$  dependencies to be broken, an overall stubbing complexity for the order  $o$  is computed as

$$OCplx(o) = \sum_{k=1}^d SCplx(k) \quad (6)$$

The principle of not breaking inheritance and composition edges was ensured by constraints in Briand et al.'s work. This paper takes a different approach, specifically, assigning higher values to the variable  $C$  in equation 1.

As shown in equation 1, our coupling measures use the number of parameters and the number of return value types in addition to the number of variables and the number of methods. We use the same normalization method as Briand et al., and include the additional coupling measures in the stubbing complexity estimation.

Using aggregated coupling measures on edges and nodes, a stubbing complexity is estimated as follows:

$$SCplx(i, j) = C + (W_V \times \overline{V}(i, j)^2 + W_M \times \overline{M}(i, j)^2 + W_R \times \overline{R}(i, j)^2 + W_P \times \overline{P}(i, j)^2)^{1/2} \quad (7)$$

where  $C$  is the first variable from equations 2 and 3,  $W_V$ ,  $W_M$ ,  $W_R$ , and  $W_P$  are weights and  $W_V + W_M + W_R + W_P = 1$ . The  $\overline{V}(i, j)$ ,  $\overline{M}(i, j)$ ,  $\overline{R}(i, j)$ , and  $\overline{P}(i, j)$  values are computed from equation 4 using values from equations 2 and 3.

Our objective is to find an optimal integration and test order  $o$  by determining a set of  $k$  nodes and/or  $l$  edges to be removed to make the *WORD* acyclic such that the sum of the stubbing complexities for these nodes and edges is minimum:

$$OCplx(o) = \sum_{i=1}^k SCplx_{node}(i) + \sum_{j=1}^l SCplx_{edge}(j) \quad (8)$$

### 3.2 Heuristics Algorithm for Breaking Cycles

This section presents a general algorithm for making a cyclic graph acyclic using simple weight assignments on edges. Algorithm 1 is illustrated through the example in Figure 1.

Figure 1 is taken from Briand et al.'s paper [2]. The edges represent general dependencies between two classes,

#### Algorithm 1 Eliminating Cycles in WORD (V,E)

---

```

1: Find all SCCs in WORD
2: for (each  $scc_i(V_{scc_i}, E_{scc_i}) \in SCCs$ ) do
3:   find all cycles CYCLES (totalCycles)
4:   for (each  $e \in E_{scc_i}$ ) do
5:     find the number of cycles that use  $e$ 
      ( $cardinal\{cycles - through - e\}$ )
6:     compute the cycle-weight ratio
7:   end for
8:   while (totalCycles != 0) do
9:     order all edges according to the descending order of
      their cycle-weight ratio
10:    remove the edge with the highest cycle-weight ratio
11:    totalCycle = totalCycle - number of cycles broken
12:    update the number of cycles that use  $e$ 
      ( $cardinal\{cycles - through - e\}$ ) in the remaining
      edge set
13:    recompute the cycle-weight ratio for the remaining
      edges
14:   end while
15: end for

```

---

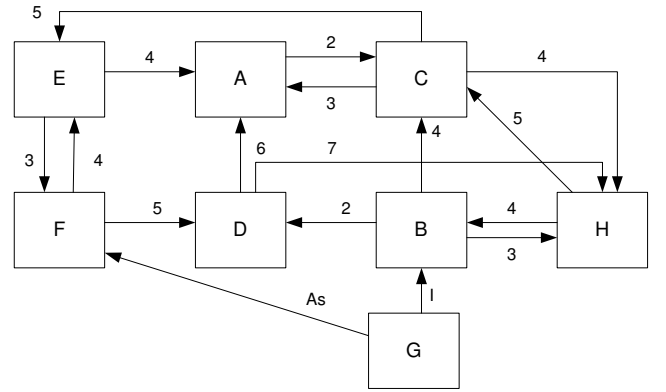


Figure 1: Example Weighted Object Relation Diagram (WORD)

not UML-specific relationships, with edge labels representing the specific stub complexity. Step 1 in Algorithm 1 finds one SCC in Figure 1,  $\{E, A, C, H, D, B, F\}$ . Steps 2 and 3 find the following 11 cycles in the *SCC*.

Table 1 shows the results from steps 4 through 7 of Algorithm 1. After the initial computation of CWR values for edges, the algorithm works in the following steps:

**A.** Choose an edge from table 1 with maximum cycle-weight ratio and remove that edge from the *WORD*. At this point, the edge with the maximum cycle-weight ratio is  $A \rightarrow C$  with a ratio of 2. Removing edge  $A \rightarrow C$  breaks four cycles, 2, 3, 6, and 7, leaving seven cycles.

**B.** Re-compute the cycle-weight ratio for the remaining edges. The result is shown in table 2. There are two edges that have same maximum cycle-weight ratio, 14 and 11 with ratios of 1. Our rule in this situation is to choose the edge that is involved in larger number of cycles. This is edge 14,  $H \rightarrow B$ . Removing  $H \rightarrow B$  breaks four cycles, 5, 9, 10, and 11, leaving three cycles.

**C.** Re-compute the cycle-weight ratio for the remaining edges in table 2. For space reasons, the result is not shown,

**Table 1: Cycle-weight Ratio for Edges in SCC  $\{E, A, C, H, B, D, F\}$**

| No. | Edge              | Wt.      | Cycles Involved     | NC       | CWR      |
|-----|-------------------|----------|---------------------|----------|----------|
| 1   | <b>A → C</b>      | <b>2</b> | <b>{2, 3, 6, 7}</b> | <b>4</b> | <b>2</b> |
| 2   | $B \rightarrow C$ | 4        | {5, 9}              | 2        | 0.5      |
| 3   | $B \rightarrow D$ | 2        | {6, 11}             | 2        | 1        |
| 4   | $B \rightarrow H$ | 3        | {10}                | 1        | 0.33     |
| 5   | $C \rightarrow E$ | 5        | {2, 3, 4, 5}        | 4        | 0.8      |
| 6   | $C \rightarrow A$ | 3        | {7}                 | 1        | 0.33     |
| 7   | $C \rightarrow H$ | 4        | {6, 8, 9}           | 3        | 0.75     |
| 8   | $D \rightarrow A$ | 6        | {3, 6}              | 2        | 0.33     |
| 9   | $D \rightarrow H$ | 7        | {4, 5, 11}          | 3        | 0.43     |
| 10  | $E \rightarrow A$ | 4        | {2}                 | 1        | 0.25     |
| 11  | $E \rightarrow F$ | 3        | {1, 3, 4, 5}        | 4        | 1.33     |
| 12  | $F \rightarrow D$ | 5        | {3, 4, 5}           | 3        | 0.6      |
| 13  | $F \rightarrow E$ | 4        | {1}                 | 1        | 0.25     |
| 14  | $H \rightarrow B$ | 4        | {5, 6, 9, 10, 11}   | 5        | 1.25     |
| 15  | $H \rightarrow C$ | 5        | {4, 8}              | 2        | 0.4      |

but the edge with maximum cycle-weight ratio is now edge 11,  $E \rightarrow F$ . Removing edge  $E \rightarrow F$  breaks two cycles, 1 and 4, leaving only cycle 8.

D. Re-compute the cycle-weight ratio for the remaining edges, and at this point the edge with maximum cycle-weight ratio is edge 7,  $C \rightarrow H$ . Removing  $C \rightarrow H$  breaks cycle 8, and makes the *WORD* acyclic.

### 3.2.1 Applying Algorithm 1 to A Special Case

A key difference between this research and previous research is the modeling of the cost of stubbing as edge weights. If we assign a weight of 1 to each edge, then our model is equivalent to the previous models. To facilitate comparison, we assign all edges in the graph in Figure 1 the weight 1, and then see if our algorithm gets the same results as Briand’s [4, 2].

Table 3 shows initial CWR values for edges. We briefly describe the process: first, edge  $H \rightarrow B$  is chosen to be removed, breaking five cycles. The re-computation of CWR values for the remaining edges are not shown, but the next edge to remove is  $E \rightarrow F$ , breaking three cycles. Next, edge  $A \rightarrow C$  is chosen, breaking two cycles. There is one cycle left, 8, and we can break it by either removing  $H \rightarrow C$  or  $C \rightarrow H$ . Here we can apply the heuristic of not to break an *Aggregation* relationship and choose  $C \rightarrow H$  to remove. Thus, we removed four edges in total:  $H \rightarrow B$ ,  $E \rightarrow F$ ,  $A \rightarrow C$ , and  $C \rightarrow H$ . This result is the same as the result from Briand et al.’s graph-based research, although the edges were removed in a different order.

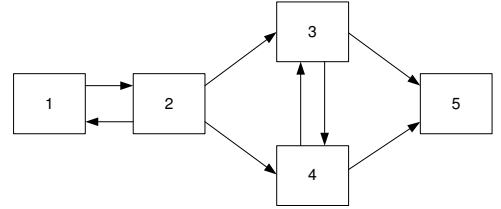
## 3.3 Algorithm for Ordering Classes for Integration Testing

Once cycles are broken by automation, the integration tester needs a specific ordering of the classes, especially for classes that appear in different SCCs. Algorithm 2 describes an approach for ordering classes for integration and testing. The algorithm first generates a *precedence table* for nodes in the *WORD*, then finds all strongly connected components (SCCs) in the weighted object relation diagram (*WORD*). A

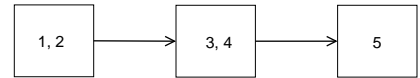
**Table 2: Cycle-weight Ratio for Edges in SCC  $\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$**

| No. | Edge              | Wt.      | Cycles Involved       | NC       | CWR      |
|-----|-------------------|----------|-----------------------|----------|----------|
| 2   | $B \rightarrow C$ | 4        | {5, 9}                | 2        | 0.5      |
| 3   | $B \rightarrow D$ | 2        | {11}                  | 1        | 0.5      |
| 4   | $B \rightarrow H$ | 3        | {10}                  | 1        | 0.33     |
| 5   | $C \rightarrow E$ | 5        | {4, 5}                | 2        | 0.4      |
| 6   | $C \rightarrow A$ | 3        | { }                   | 0        | 0        |
| 7   | $C \rightarrow H$ | 4        | {8, 9}                | 2        | 0.5      |
| 8   | $D \rightarrow A$ | 6        | { }                   | 0        | 0        |
| 9   | $D \rightarrow H$ | 7        | {4, 5, 11}            | 3        | 0.43     |
| 10  | $E \rightarrow A$ | 4        | { }                   | 0        | 0        |
| 11  | <b>E → F</b>      | <b>3</b> | <b>{1,4,5}</b>        | <b>3</b> | <b>1</b> |
| 12  | $F \rightarrow D$ | 5        | {4, 5}                | 2        | 0.4      |
| 13  | $F \rightarrow E$ | 4        | {1}                   | 1        | 0.25     |
| 14  | <b>H → B</b>      | <b>4</b> | <b>{5, 9, 10, 11}</b> | <b>4</b> | <b>1</b> |
| 15  | $H \rightarrow C$ | 5        | {4, 8}                | 2        | 0.4      |

*precedence table* is indexed by the number or name of nodes in the *WORD*, and shows the nodes that are connected to a node through outgoing edges from the node. Then, each SCC is compressed into a node, and the multiple edges between SCCs are combined into one edge. As a result, an acyclic directed graph  $WORD_{comp}$ , a compressed *WORD*, is produced. For example, Figure 2 represents a *WORD* with three SCCs, {1,2}, {3,4}, and {5}. Figure 3 shows the resulting  $WORD_{comp}$ . The algorithm finds the reversed topological order,  $O_{SCCs}$ , for the  $WORD_{comp}$  as (1) {5}, (2) {3,4}, and (3) {1,2}. Then, each  $scc_i$  is made acyclic



**Figure 2: Finding Overall Test Order in a *WORD***



**Figure 3: Compressed *WORD***

using Algorithm 1, and let  $scc_{i-acyclic}$  represent the resulted subgraph. The removed edges represent specific test stubs to be developed. Suppose edges  $1 \rightarrow 2$  and  $3 \rightarrow 4$  are removed from SCCs {1,2} and {3,4}. The result is that there are two specific test stubs for 2 and 4. A reverse topological order,  $O_{scc_i}$ , is generated for  $scc_{i-acyclic}$ . In this example, SCCs {1,2} and {3,4} have reverse topological order of 1, 2 and 3, 4. Testing starts according to the order  $O_{SCCs}$ . For each node in  $O_{SCCs}$ , first,  $scc_{i-acyclic}$  is restored, and included nodes are tested according to the order  $O_{scc_i}$ . For

**Table 3: Cycle-weight Ratio for Edges in SCC  $\{E, A, C, H, B, D, F\}$  - All Edges Have the Same Weight**

| No. | Edge              | Wt. | Cycles Involved   | NC | CWR |
|-----|-------------------|-----|-------------------|----|-----|
| 1   | $A \rightarrow C$ | 1   | {2, 3, 6, 7}      | 4  | 4   |
| 2   | $B \rightarrow C$ | 1   | {5, 9}            | 2  | 2   |
| 3   | $B \rightarrow D$ | 1   | {6, 11}           | 2  | 2   |
| 4   | $B \rightarrow H$ | 1   | {10}              | 1  | 1   |
| 5   | $C \rightarrow E$ | 1   | {2, 3, 4, 5}      | 4  | 4   |
| 6   | $C \rightarrow A$ | 1   | {7}               | 1  | 1   |
| 7   | $C \rightarrow H$ | 1   | {6, 8, 9}         | 3  | 3   |
| 8   | $D \rightarrow A$ | 1   | {3, 6}            | 2  | 2   |
| 9   | $D \rightarrow H$ | 1   | {4, 5, 11}        | 3  | 3   |
| 10  | $E \rightarrow A$ | 1   | {2}               | 1  | 1   |
| 11  | $E \rightarrow F$ | 1   | {1, 3, 4, 5}      | 4  | 4   |
| 12  | $F \rightarrow D$ | 1   | {3, 4, 5}         | 3  | 3   |
| 13  | $F \rightarrow E$ | 1   | {1}               | 1  | 1   |
| 14  | $H \rightarrow B$ | 1   | {5, 6, 9, 10, 11} | 5  | 5   |
| 15  | $H \rightarrow C$ | 1   | {4, 8}            | 2  | 2   |

Figure 2, the integration and test order is 5, 3, 4, 1, 2. Before a node is tested, the *precedence table* is checked. If a node was connected to a removed edge, then include the corresponding test stub in the test order. For example, when node 3 is tested, the precedence table indicates that node 3 is connected to an untested node. Thus, the test stub for node 4 is included at this point.

**Algorithm 2 Ordering Classes for Integration & Testing**

- 1: Generate *precedence table* for nodes in the WORD
- 2: Find all SCCs in WORD
- 3: Generate an acyclic compressed version of WORD,  $WORD_{comp}$ , by representing each  $scc_i$  as a single node and by compressing multiple edges between every two nodes
- 4: Find reverse topological order,  $O_{SCCs}$ , of nodes in  $WORD_{comp}$
- 5: **for** (each  $scc_i \in SCCs$ ) **do**
- 6:   make  $scc_i$  acyclic by using Algorithm 1 and record removed edges
- 7:   find reverse topological order,  $O_{scc_i}$ , for nodes in the acyclic  $scc_i$ -*acyclic*
- 8: **end for**
- 9: start testing according to the order of SCCs in  $O_{SCCs}$
- 10: **for** (each ordered  $scc_i \in SCCs$ ) **do**
- 11:   test nodes in the order  $O_{scc_i}$
- 12: **end for**

**4. CASE STUDY**

This section provides a preliminary evaluation of the model and algorithms by comparing results with the same project, the ATM system, used by Briand et al. [2]. Briand et al. chose the number of broken dependencies, attribute couplings, method couplings, and a combination of attributes and methods as four cost functions to produce an integration

test order, and compared the results to decide which cost function gives the best result. Our approach is first to use dependencies, attribute coupling measures, method coupling measures, and a combination of attribute and method coupling measures as weights on edges and apply our algorithm to check what kind of result can be obtained under similar situations. Then, we collect coupling data from the implementations using coupling definitions and coupling measures defined in Section 3.1, construct the weighted object relation diagram (WORD), and compute the edge weights for SCCs in the WORD using equations 2 and 7.

The ATM system has 21 classes and eight form a strongly connected component that has 30 cycles [2]. Table 4 shows the coupling measures in the format of equation 1. Table 5 shows the different edge weights that are used in this evaluation. In particular, the columns labeled Dependency, # of Attributes, # of Methods, and A & M show the edge weights obtained from Briand et al.'s four cost functions. The last column shows edge weights that are computed from Table 4 using equations 4, 2, 3, and 7. The constraints of not breaking inheritance and composition edges are achieved by assigning 5 to variable  $C$  in equation 1 for inheritance and composition, and 1 for the others.

**Table 5: Different Weights for Edges in SCC  $\{8, 9, 10, 11, 12, 13, 14, 15\}$**

| No. | Edge                | Dependency | # of Attr. | # of Meth. | A & M    | A & M-new |
|-----|---------------------|------------|------------|------------|----------|-----------|
| 1   | $8 \rightarrow 9$   | 1          | 13         | 1          | 0.71     | 1         |
| 2   | $8 \rightarrow 10$  | 1          | 9          | 2          | 0.53     | 1.22      |
| 3   | $9 \rightarrow 8$   | 1          | 13         | 7          | 1        | 1.17      |
| 4   | $10 \rightarrow 8$  | 1          | 13         | 7          | 1        | 1.66      |
| 5   | $10 \rightarrow 9$  | 1          | 13         | 2          | 0.74     | 1.13      |
| 6   | $10 \rightarrow 11$ | 1          | 0          | 0          | 0        | 1.57      |
| 7   | $10 \rightarrow 12$ | 1          | 2          | 2          | 0.23     | 1.13      |
| 8   | $10 \rightarrow 13$ | 1          | 2          | 2          | 0.23     | 1.13      |
| 9   | $10 \rightarrow 14$ | 1          | 3          | 2          | 0.26     | 1.13      |
| 10  | $10 \rightarrow 15$ | 1          | 1          | 2          | 0.21     | 1.13      |
| 11  | $11 \rightarrow 8$  | 1          | 13         | 2          | 0.74     | 1.17      |
| 12  | $11 \rightarrow 9$  | 1          | 13         | 1          | 0.71     | 1.00      |
| 13  | $11 \rightarrow 10$ | 1          | 9          | 2          | 0.53     | 1.13      |
| 14  | $12 \rightarrow 8$  | 1          | 13         | 4          | 0.81     | 1.60      |
| 15  | $12 \rightarrow 9$  | 1          | 13         | 4          | 0.81     | 2.59      |
| 16  | $12 \rightarrow 10$ | 1          | 9          | 2          | 0.53     | 2.01      |
| 17  | $12 \rightarrow 11$ | 1          | $\infty$   | $\infty$   | $\infty$ | 5.00      |
| 18  | $13 \rightarrow 8$  | 1          | 13         | 4          | 0.81     | 1.50      |
| 19  | $13 \rightarrow 9$  | 1          | 13         | 4          | 0.81     | 2.62      |
| 20  | $13 \rightarrow 10$ | 1          | 9          | 2          | 0.53     | 2.01      |
| 21  | $13 \rightarrow 11$ | 1          | $\infty$   | $\infty$   | $\infty$ | 5.00      |
| 22  | $14 \rightarrow 8$  | 1          | 13         | 3          | 0.77     | 1.39      |
| 23  | $14 \rightarrow 9$  | 1          | 13         | 3          | 0.77     | 2.58      |
| 24  | $14 \rightarrow 10$ | 1          | 9          | 2          | 0.53     | 2.01      |
| 25  | $14 \rightarrow 11$ | 1          | $\infty$   | $\infty$   | $\infty$ | 5.00      |
| 26  | $15 \rightarrow 8$  | 1          | 13         | 2          | 0.74     | 1.29      |
| 27  | $15 \rightarrow 9$  | 1          | 13         | 3          | 0.77     | 2.58      |
| 28  | $15 \rightarrow 10$ | 1          | 9          | 2          | 0.53     | 2.01      |
| 29  | $15 \rightarrow 11$ | 1          | $\infty$   | $\infty$   | $\infty$ | 5.00      |

In all five approaches, seven dependencies were removed.

Table 4: Coupling Measures for Edges in *SCC* {8, 9, 10, 11, 12, 13, 14, 15}

| No. | 8          | 9          | 10        | 11        | 12        | 13       | 14        | 15        |
|-----|------------|------------|-----------|-----------|-----------|----------|-----------|-----------|
| 8   |            | 1.0.0.0.0  | 1.0.2.1.3 |           |           |          |           |           |
| 9   | 1.0.1.1.3  |            |           |           |           |          |           |           |
| 10  | 1.0.6.4.4  | 1.0.1.1.1  |           | 1.0.3.5.3 | 1.0.1.1.0 | 10.1.1.0 | 1.0.1.1.0 | 1.0.1.1.0 |
| 11  | 1.0.1.1.3  | 1.0.0.0.0  | 1.0.1.1.0 |           |           |          |           |           |
| 12  | 1.0.4.3.11 | 1.0.4.3.11 | 1.0.2.2.0 | 5.0.0.0.0 |           |          |           |           |
| 13  | 1.0.4.3.6  | 1.0.4.3.14 | 1.0.2.2.0 | 5.0.0.0.0 |           |          |           |           |
| 14  | 1.0.3.2.6  | 1.0.3.3.12 | 1.0.2.2.0 | 5.0.0.0.0 |           |          |           |           |
| 15  | 1.0.2.1.6  | 1.0.3.3.10 | 1.0.2.2.0 | 5.0.0.0.0 |           |          |           |           |

When using weights in the columns labeled # of Attributes, # of Methods, A & M, and A & M-new of Table 5, exactly same set of edges were removed. Hence, the stubbing cost for these approaches are equal. Although seven dependencies were broken when using the existence of dependencies as a cost function, the stubbing cost may vary because the edge weights are the same and thus cannot reflect any stubbing cost at the time of deciding to choose an edge to remove between two equal weight edges.

The results indicate that when we consider the stub complexity as weights on edges, graph-based algorithms can produce results as good as those produced by genetic algorithms. A larger empirical evaluation needs to be carried out to verify that this result generalizes.

## 5. CONCLUSIONS AND FUTURE WORK

This paper presents an improved technique and algorithms to automate the CITO problem. The technique uses weights to represent the cost of creating stubs. This has been done before, but the weights in this research are derived from quantitative analysis of couplings, thus obtaining more precise results. These weights are placed on a *Weighted Object Relation Diagram (WORD)*, which represents classes as nodes and relationships as edges. This paper also introduces the idea of applying weights to nodes to estimate the cost of removing the nodes. If a class is used by multiple classes, then all or part of the same stub for that class may be shared among all classes that use it, thus reducing the cost of stubbing. The weight of a node is at least as high as the maximal weight of all incoming edges (assuming total sharing of the stub), and no higher than the sum of the weights of all incoming edges (assuming no sharing of the stub).

A new algorithm to solve the CITO problem is introduced. This algorithm uses edge weights; the part that uses node weights is abbreviated to conserve space. The algorithm was compared with algorithms by previous researchers, and found to be just as effective if edge weights are ignored. Overall, the results in this paper improve the ability for developers to automate the CITO problem. An open question is how well these results will scale to large systems and we hope to work on that problem soon. In the future, we plan to complete automation of this work and then carry out detailed experiments to fully assess the value of the technique.

## 6. ACKNOWLEDGMENTS

We would like to thank Prof. Lionel Briand and his colleagues for sharing their experimental materials with us.

## 7. REFERENCES

- [1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.
- [2] L. Briand, J. Feng, and Y. Labiche. 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*, pages 43–50, Ischia, Italy, 2002. IEEE Computer Society Press.
- [3] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. Technical report SCE-01-02, Carleton University, 2001.
- [4] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, July 2003.
- [5] M. J. Harrold and J. D. McGregor. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80, Melbourne, Australia, May 1992. IEEE Computer Society Press.
- [6] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *19th Computer Software and Applications Conference (COMPSAC '95)*, pages 239–244, Dallas, TX, August 1995. IEEE Computer Society Press.
- [7] B. A. Malloy, P. J. Clarke, and E. L. Lloyd. 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)*, pages 353–364, Denver, Colorado, 2003. IEEE Computer Society Press.
- [8] K.-C. Tai and F. Daniels. Test order for inter-class integration testing of object-oriented software. In *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 602–607, Santa Barbara CA, 1997. IEEE Computer Society.
- [9] Y. L. Traon, T. Jéron, J.-M. Jézéquel, and P. Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, pages 12–25, March 2000.