Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles

Lionel C. Briand, Yvan Labiche, Yihong Wang
Software Quality Engineering Laboratory
Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada
{briand, labiche, yihong}@sce.carleton.ca

ABSTRACT
The issue of ordering class integration in the context of integration testing has been discussed by a number of researchers. More specifically, strategies have been proposed to generate a test order while minimizing stubbing. Recent papers have addressed the problem of deriving an integration order in the presence of dependency cycles in the class diagram. Such dependencies represent a practical problem as they make any topological ordering of classes impossible. Three main approaches, aimed at “breaking” cycles, have been proposed. The first one was proposed by Tai and Daniels and is based on assigning a higher-level order according to aggregation and inheritance relationships and a lower-level order according to associations. The second one was proposed by Le Traon et al and is based on identifying strongly connected components in the dependency graph. Among other things, the former approach may result into unnecessary stubbing whereas the latter may lead to breaking cycles by “removing” aggregation or inheritance dependencies, thus leading to complex stubbing. As a result, a third approach, that combines some of the principles of both approaches and addresses some of their shortcomings, was proposed. This paper reviews these strategies (principles are described, advantages and drawbacks are precisely investigated), and provides both analytical and empirical comparisons based on 5 case studies.
Keywords
Integration testing, Integration order, Object-Oriented Software Engineering, Experimentation

1 INTRODUCTION
One important problem when integrating and testing object-oriented software is to decide the order of class integration [2]. A number of papers have provided strategies and algorithms to derive an integration and test order from dependencies among classes in the system class diagram [5, 12, 13, 15, 21]. The objective of all these approaches is to minimize the number of test stubs to be produced, as this is perceived to be a major cost factor of integration testing. Indeed, stubs are pieces of software that have to be built (cost) in order to simulate parts of the software that are either not developed yet or still to be unit tested, but are needed to test classes that depend on them. Kung et al were the first ones 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 know graph theory problem¹. In the presence of dependency cycles, the proposed strategy consists in identifying strongly connected components (SCCs) and removing associations until there is no cycle in the SCCs. However, they do not provide precise solutions when there is more than one candidate association for cycle breaking (one is randomly selected): They only mention that a possible solution would involve the use of the complexity of the associations involved in cycles. But such dependency cycles are commonplace in real world systems and hinder any topological sorting of classes. Cycles are usually rare in Analysis class diagrams but then it is common, as design progresses, to add classes and relationships in order, 

¹ Topological sorting of a graph G consists in numbering the vertices of G (e.g., labeling them with numbers 1, …, n) such that for each edge, the label associated to the source vertex is strictly lower than the label associated to the target vertex. Furthermore, it can be shown that there is a topological sorting for any directed acyclic graph [10].
for example, to improve performance or maintainability. As a result, the class diagram usually contains cycles by the end of low-level design.

All solutions are based on the principle of “breaking” some dependencies to obtain acyclic dependencies between classes. In our context, a broken dependency implies that the target class will have to be stubbed when integrating and testing the source class. Tai and Daniels [21] proposed a 2-stage algorithm that deals with dependency cycles. However, in cases where class associations are not involved in cycles, their solution is sub-optimal in terms of the required number of test stubs. Le Traon et al [15] subsequently proposed an alternative strategy that is based on graph search algorithms to recognize strongly connected components and that arguably yields more optimal results. One issue though, is that this algorithm is not fully deterministic in the sense that, depending on some arbitrary decision (e.g., the initial vertex (class) of the search, and the search itself), the algorithm may yield significantly different results. Furthermore, since the model used does not have any information on the kind of dependency (inheritance, association or aggregation) this approach may lead to the removal of an inheritance or aggregation relationship. Kung et al [12], as well as others before them [20], pointed out that association relationships are usually the weakest links in a class diagram, i.e., the links involving the least dependencies and therefore the least stub complexity if broken. Kung et al further argued that every cycle in a class diagram contains at least one association. Though no demonstration is provided in their paper, this is due to the fact that inheritance and aggregation relationships are defined as transitive, irreflexive, and asymmetrical [17]. It is then easy to show that a cycle involving only aggregation and inheritance relationships would lead, assuming transitivity, to symmetrical or reflexive dependencies between instances, thus transgressing one of the basic properties of these relationships. As a simple example, if classes A and B are related
through compositions with B and C, respectively, and C inherits from A, then A instances are composed of C instances (transitivity), which are themselves A instances thus transgressing the irreflexivity property. A third approach, based on the ones presented above and that addresses important weaknesses, was then proposed [5]. It also identifies SCCs with a graph-based algorithm, but differs in the way cycles are broken (e.g., it does not remove inheritance or aggregation relationships).

It is worth noting that other approaches have recently been proposed [4, 14]. They are no longer graph-based but rather make use of Genetic Algorithms (GA’s), a global optimization technique based on heuristics, which has been developed by the artificial intelligence community over the years. Though this is a promising avenue of research in order to facilitate the use of more complex stubbing complexity definitions [4], we do not investigate such an approach in this article, as we want to focus on graph-based techniques which are simpler to apply, more efficient to use, and as GA’s involve a whole new set of challenges [4].

Last, there exist other integration strategies that are not based on the class diagram, as derived from the software design or reverse-engineering. They rather associate a functional description with (a set of) classes. For instance, in [9], Atomic System Functions (ASF), which involve system inputs and outputs, and exercise Method/Message paths between objects, drive the integration test of classes. These ASFs correspond to a functional decomposition of the system, which is similar to use cases. The objective of the strategy is not to minimize test stubs but to execute complete, end-user functionalities, in an incremental manner during integration. Similar strategies using use cases can be found in [2, 18]. Since these strategies are not explicitly based of the class diagram, they will not be detailed and compared in this article. Though this is the
topic of future research, it is very likely that in practice, those two sets of strategies would have to be combined.

Section 2 first introduces some important issues in assessing integration ordering strategies. In Section 3, we present a critical analysis of existing strategies, thus pondering their strengths and drawbacks using an example. Section 4 presents an empirical evaluation of the three approaches using five different application systems, and conclusions are drawn in Section 5.

2 PRELIMINARY ISSUES IN ASSESSING INTEGRATION ORDERS

Before going into the depth of integration order strategies, a number practical issues need to be clarified.

2.1 Classes Stubbed Versus Stubs

One important question is to determine what should be the criterion to evaluate strategies that break cycles. One possibility, which is implied by both Tai and Daniels and Le Traon et al, is to count the number of classes to be stubbed. However, when a client class uses a stub to be tested, this stub usually emulates the minimal subset of the server class functionality that is required for that specific client. In other words, stubs need to remain as simple as possible (e.g., in terms of control and data flow) so as not to become error-prone: “If the stub is realistic in every way, it is no longer a stub but the actual routine.” [1] Ideally, stubs should only contain sequential control flow so as to require little testing effort themselves. Therefore, when a server class is used by several client classes, we usually obtain at least as many stubs as client classes, if we want to minimize risks. It is in any case more likely that the number of stubs will be proportional to the number of client classes of classes to be stubbed, as opposed to just the latter ones. This leads us to the position that the number of stubbed classes multiplied by their client classes is probably a more realistic evaluation criterion than the number of stubbed classes alone. Le Traon et al [15]
refers to this number as \textit{specific} stubs and we reuse this term below. They, however, refer to the number of stubbed classes as \textit{realistic} stubs. Since, as discussed above, a realistic number of stubs is more likely proportional to the number of client classes of classes to be stubbed, we choose not to use this term any further.

However, the number of specific stubs still remains an estimate of the cost of a class test order as not all broken dependencies lead to stubs of similar complexity. This simplifying assumption is used by all the algorithms referenced above and may seem simplistic, but our observation has been that, in terms of number of methods and attributes in the target class, frequency distributions are similar across test orders so that orders entailing larger numbers of stubs are still likely to be more expensive than orders with lower numbers of stubs.

When assessing the stubbing cost involved in a test order, one may want to look further than the number of stubs so as to get more precise indicators of the cost of producing stubs. In order to increase the precision of our evaluations across our case studies, we not only show the number of stubs but also what they correspond to in terms of methods and attributes (from the target class of the broken dependency) that may potentially be involved in the stubs. Though these two indicators are admittedly not perfect, they nevertheless improve the precision of the results reported.

\section*{2.2 Stubby Associations Versus other Relationships}

As mentioned above (and explained in Section 3.2), Le Traon et al [15] may lead to “breaking” aggregation or inheritance relationships, thus leading to the stubbing of these relationships. If we take, as an example the state design pattern [7], the \textit{context} class whose state is being modeled is related, through an aggregation relationship, to an abstract class \textit{State} (Figure 1). Every time the context class receives a message, its state is likely to change, thus requiring a message being sent
to an instance of one of the subclasses of State. In turn, the subclasses instances may invoke action methods in the context class. So we see the dependencies are very tight and we claim that this is usually the case with aggregation relationships. Regarding inheritance, stubbing a parent class would imply stubbing most inherited methods. This follows from the fact that inherited methods should, in many instances, be tested in the new context represented by a subclass, even though they may have been fully tested in the parent class. In [19] the authors show, in the light of the adequacy criterion defined in [24], that inherited code need to be retested. Furthermore, Harrold and McGregor [8] propose an incremental strategy for testing inheritance hierarchies that minimizes re-testing of inherited methods. This strategy requires that inherited methods already tested in the parent class A be retested in the child class B only if they interact with B’s methods. In practice, when inheritance is properly used to extend the functionality of parent classes (Liskov principle [16]), most parent class methods inherited are usually interacting with at least one child class member. As a result, when breaking inheritance relationships, the resulting stubs would almost have to be the entire parent class, as all inherited, non-overwritten methods would have to be tested in the subclass to consider it unit tested. So, we clearly see from the discussion above that any method breaking cycles should aim at only breaking association relationships, so as to require the development of stubs that are economically viable.

![Figure 1 State Design Pattern](image-url)
It ensues that the algorithms we are about to describe need a model of class relationships that allows the identification of breakable and unbreakable relationships. Such models have been defined and used in previous works [11, 21]: The Object Relation Diagram (ORD) identifies inheritance (I), aggregation (Ag), and association (As) relationships, the latter being the only breakable relationship [12] (a mapping from UML to the ORD has been proposed in [5]); The Test Dependency Graph (TDG) extends the ORD by including relationships at the method level [14, 15]. However, the mapping from UML to the TDG proposed in [14, 15] does not distinguish UML aggregation and composition relationships, making the latter breakable. Since we do not need information at the method level when devising a class test order and we do not want to break composition relationships, we will use the ORD and the mapping proposed in [5] in the remainder of this article.

3 EXISTING GRAPH-BASED STRATEGIES

We follow below a chronological order to present existing work. The following running example (see the ORD in Figure 2) is used to illustrate the techniques and discuss their implications. It is a modified version of the example used in [21]. Classes are labeled with capital letters and dependencies are labeled according to their type: As for associations, Ag for aggregations, I for inheritance. Though this example may seem a bit complex and dense, it is aimed at supporting our argumentation using a minimum number of classes and dependencies. The case studies in Section 4 will present real examples.

Note that Kung et al.’s approach does not appear in the comparison below, as Kung et al.’s approach consists in a random selection of associations within SCCs (Section 1) whereas the other strategies provide more enhanced strategies for the selection of relationships to break.
3.1 Tai and Daniels

Tai and Daniels [21] proposed a strategy that assigns each class in the class diagram a major and a minor level number. Those numbers are then used to devise an integration order. Major level numbers are assigned based on inheritance and aggregation dependencies only: since all association dependencies are ignored, there is no cycle and topological sorting can be applied. Then, minor level numbers are assigned, within each major level, based on association dependencies only. Here cycles may appear and must be broken in order to apply topological sorting. In that case, a $\text{weight}(d_i)$ function is defined for each association $d_i$ in each major level as the number of the incoming dependencies of the origin node of $d_i$ plus the number of outgoing dependencies of the target node of $d_i$. The rationale is that the higher the weight the more likely breaking a dependency will break a larger number of cycles. As we will see below, this is only a simple heuristic, not a guarantee. Dependencies with higher weights are therefore selected to break cycles. For the reasons discussed in Section 2.2, only associations are broken, as other types of dependencies would more likely lead to more complex stubs.

3.1.1 Application on the example

Using the example above, we derive the major level and minor level numbers for each class, using the algorithms provided in [21]. Figure 3.a shows the ORD considered for the determination of major levels: the ORD is derived from Figure 2 and only contains aggregation dependencies.
and inheritance relationships. There is no cycle and topological sorting assigns major levels: 1 to classes E, A, and C; 2 to classes F and H; 3 to classes D and B; 4 to class G. Then Figure 3.b shows classes at major level 1 (the only major level containing a cycle when considering associations) and their association relationships so as to determine their minor level numbers. The weights of the different associations are: \( W(C, A) = 2; W(E, A) = 2; W(A, C) = 4, W(C, E) = 2 \). Association \((A, C)\) is broken and classes A, E, and C have minor level numbers 1, 2, and 3, respectively.

\[
\begin{align*}
\text{(a) Figure 2 without associations for major level numbers} \\
\text{(b) Associations inside major level 1 for minor level numbers}
\end{align*}
\]

\text{Figure 3 Intermediate steps of Tai and Daniels’ approach for Figure 2}

Figure 4 shows the example classes sorted according to their major level, from top to bottom, and indicates the minor level numbers within the class boxes, denoted as decimal numbers, e.g., class F has major level 2 and minor level 1, leading to level 2.1. Also visible on Figure 4 are the dependencies that are broken to obtain the final order, thus leading to five classes to be stubbed (B, C, D, F, H) and five specific stubs, that is one for each class. The right part of Figure 4 shows the final test order.
3.1.2 Comments

With Tai and Daniels, associations that cross major levels in an ascending way\(^3\) (source and target classes are in major levels \(i\) and \(j\) respectively, such that \(i < j\)) are systematically broken, thus leading to stubs. Other associations that cross major levels (in a descending way) do not lead to stubs because the target class is tested before the source class (descending major levels).

In Figure 4, 4 associations are broken as they cross major levels in an ascending way (e.g., association \((E, F)\)). However, since association \((E, F)\) is deleted and \(D\) and \(F\) are no more involved in any cycle, stubbing \(D\) is a priori not necessary and is just an artifact of Tai and Daniel’s algorithm. Through this example we see how their algorithm can lead to suboptimal solutions when there are associations crossing major levels that are not involved in dependency cycles\(^4\). There are two distinct cases: The association was never involved in any cycle or, because other associations are deleted, the cycles they were originally involved in are already broken, the latter case being the situation we encounter here.

---

\(^2\) Stub(C, A) denotes any stub of C fulfilling the needs for testing A.

\(^3\) Though in Figure 4, from a purely graphical point of view, they appear in a descending way.

\(^4\) Note that, this problem could be solved by identifying associations that are not part of any cycle before the application of the approach, and considering these associations as aggregation, i.e., they are not part of the set of associations that are removed during the first step.
3.2 Le Traon et al

Le Traon et al use a very different strategy to deal with dependency cycles [15]. This approach identifies SCCs using an adaptation of Tarjan’s algorithm [22]. It is adapted such that each dependency in the considered graph is labeled, as per the order of traversing, according to a classification scheme. One type of dependency, which is used in the decision to break cycles, is denoted as \textit{frond} dependency. It is defined as going from a vertex (class) to one of its “ancestors”, i.e., a vertex that is traversed before in the depth-first search or, in other words, a class that depends on it, directly or indirectly.

Le Traon et al breaks the cycles by removing the incoming dependencies of the class with the highest \textit{weight}, in the considered SCC. The weight here is defined differently from that of Tai and Daniels: It is the sum of incoming and outgoing \textit{frond} dependencies for a given class, within the SCC under consideration. In short, the notion of weight is defined on classes and it specifically focuses the notion of frond dependency (which capture some of the cycles in which the class is involved). For each non-trivial SCC (with more than one vertex), the procedure above is then called recursively.

3.2.1 Application on the example

Since the weight is computed according to frond edges, and frond edges depend on the construction of the depth-first search (DFS) tree, the weight depends on the vertex from which we start the DFS algorithm. In this section we choose vertex G.

While applying Tarjan’s Algorithm on our example using G as a starting vertex, we find one SCC made up of classes F, E, C, A, D, B, and H (Figure 5.a). The (partial) test order is:

1. SCC\{E, A, C, F, D, B, H\} is tested;
2. G is tested using F and B.
The weight value of each vertex in the SCC are: \( W(E) = 1 \), \( W(A) = 1 \), \( W(C) = 2 \), \( W(B) = 2 \), \( W(H) = 2 \), \( W(D) = 1 \), \( W(F) = 1 \). Three vertices maximize the weight, leading to three choices for the set of broken dependencies. We decide here not to show all the three choices, but only one. B is selected and its incoming dependencies in the SCC are broken, i.e., association between H and B. Tarjan’s Algorithm is applied and identifies a lower SCC, involving classes E, A, C, F, D, and H (Figure 5.b). The (partial) test order is:

1. SCC\{E, A, C, F, D, B, H\} is tested;
   1.1 SCC\{E, A, C, F, D, H\} is tested using stub(B, H);
   1.2 B is tested using D, C and H;
2. G is tested using F and B.

(a) First call to Tarjan’s algorithm  (b) Second call to Tarjan’s algorithm

Figure 5 Le Traon’s approach for Figure 2, starting with node G (intermediate results)

We do not further detail the application of the algorithm on the example, as the above description is sufficient for our purpose (see Section 3.2.2). However, the interested reader who wishes to continue the process will see that when selecting C and F in the subsequent steps, we obtain Figure 6 which shows the identified SCCs and the corresponding broken dependencies, as well as the final test order. This solution leads to 3 stubbed classes (classes C, B, and F) and 4 specific stubs (2 for class C, 1 for class B, and 1 for class F), and breaks the aggregation between H and C.
3.2.2 Comments

From Figure 6 we see what are the frond dependencies (according to the traversal), what nested SCCs have been detected using Tarjan’s algorithm recursively, and what dependencies were deleted. The number of classes stubbed and the number of specific stubs are 3 and 4, respectively. The results are therefore better than what is obtained with Tai and Daniels (5 classes stubbed and specific stubs). However, we can see that an aggregation gets deleted (between H and C), which goes against our basic principle of exclusively breaking association dependencies (Section 2.2).

The algorithm proposed by Le Traon et al is not deterministic in the sense that the output depends on a number of arbitrary decisions. There are two levels of non-determinism. First, the result depends on the initial class where you decide to start your depth-first-search since in Tarjan’s algorithm, whether dependencies are classified as fronds or not depends on how the graph is traversed, and the weight considers only frond dependencies. Second, the algorithm does not specify what to do when classes show the same weight. In other words, depending on the graph traversal, you can obtain a different set and/or number of specific stubs and classes to be stubbed. For example, we could have selected H instead of G as the initial class for the depth-first search. Then, three choices would have been possible for the next class to traverse: A, E, F.
have the same weight. The results would then range from 6 to 7 specific stubs (instead of 4 above!), with a constant of 3 classes to stub (see Table 1).

3.3 Briand et al.

Our strategy resembles Le Traon et al since it is based on a recursive call to Tarjan’s algorithm. But it exhibits one important difference with Le Traon et al: Similar to Tai and Daniels we use a weight definition that characterizes associations by computing an estimate of the number of cycles in which the association is involved in an SCC [5].

As for Le Traon et al approach, we recursively identify SCCs using Tarjan’s algorithm. At each step, i.e., inside each non-trivial SCC, we calculate the weight of each association dependency using a modified version of Tai and Daniels’ definition, and then break the association dependency with the highest weight. Note that, as opposed to Le Traon et al’s approach, our strategy does not have the first level of non-determinism since the algorithm for computing SCCs has no effect on the weight: We do not use any dependency classification such as frond. Moreover, regarding the second level of non-determinism (alternative choices when equals weights), as opposed to Le Traon et al’s definition of weight, all alternative choices will lead to the same number of specific stubs. This is due to the fact that our criteria leads to the removal of one and only one association, whereas the criteria used by Le Traon et al lead to the removal of every incoming dependency of the selected class.

3.3.1 Weight Computation

Recall from Section 3.1 that the weight associated with each association using Tai and Daniels’s definition is the number of incoming edges of the source class plus the number of outgoing edges of the target class. The problem with this definition is that, though this is a reasonable heuristic, the number computed does not quantify anything meaningful. In our case we choose to multiply
the number of incoming and outgoing edges within the SCC under consideration. What we obtain is an estimate of the *minimum* number of cycles in which the association is involved, within a SCC\(^5\). Ideally, we would like to delete first the associations that are involved in the largest number of cycles in order to minimize stubs. Since we do not have such a count, we use as a heuristic the minimum estimated number, as per our weight definition, to select the next association to be broken in the SCC. The implications of our weight definition are as follows:

- Because our selection unit is the *association*, as opposed to the *class* for Le Traon, we can select the smaller number of associations, one by one, that breaks the largest number of cycles and really minimizes the number of *specific* stubs. On the other hand, Le Traon et al will minimize the number of classes to be stubbed, though aggregations and inheritance dependencies may be broken as a result.

- Because we use the *minimum* number of cycles in which an association is involved, our heuristic is more precise than the definition of Tai and Daniels, which does not clearly relate with the actual number of cycles the association is involved in.

### 3.3.2 Application on the example

We now apply our algorithm to the example ORD of Figure 2. First, we get the result of the first call to Tarjan’s algorithm that identifies first level SCCs in the ORD. Figure 7.a shows one non-trivial SCC: \{F, E, C, A, D, B, H\}. This implies the following (partial) test order:

1. SCC\{F, E, C, A, D, B, H\} is tested;
2. G is then tested using SCC\{F, E, C, A, D, B, H\}.

In order to continue the execution of our algorithm, we calculate each Association’s weight within SCC\{F, E, C, A, D, B, H\}:

\(^5\)Note that we could compute the actual number of cycles an edge is involved in, but this increases significantly the time complexity of the algorithm. It is a trade-off that should be explored in any specific situation.
\[ W(H, B) = H_{in} \times B_{out} = 3 \times 3 = 9 \]
\[ W(B, D) = B_{in} \times D_{out} = 1 \times 2 = 2 \]
\[ W(B, C) = B_{in} \times C_{out} = 1 \times 3 = 3 \]
\[ W(A, C) = A_{in} \times C_{out} = 3 \times 3 = 9 \]
\[ W(C, A) = C_{in} \times A_{out} = 3 \times 1 = 3 \]
\[ W(C, E) = C_{in} \times E_{out} = 3 \times 2 = 6 \]
\[ W(E, A) = E_{in} \times A_{out} = 2 \times 1 = 2 \]
\[ W(E, F) = E_{in} \times F_{out} = 2 \times 2 = 4 \]
\[ W(F, D) = F_{in} \times D_{out} = 1 \times 2 = 2 \]
\[ W(C, H) = C_{in} \times H_{out} = 3 \times 2 = 6 \]

Associations \((A, C)\) and \((H, B)\) have the same maximal weight value. We choose here \((A, C)\)\(^6\) to be deleted and apply Tarjan’s algorithm on \(SCC\{F, E, C, A, D, B, H\}\) (see Figure 7.b):

1. \(SCC\{F, E, C, A, D, B, H\}\) is tested;
   1.1 \(A\) is tested using stub\((C, A)\);
   1.2 \(SCC\{F, E, C, D, B, H\}\) is tested using \(A\) (for classes \(E, C,\) and \(D\));
2. \(G\) is tested using \(F\) and \(B\) in the previous SCC.

The weight of every association in \(SCC\{F, E, C, D, B, H\}\) is determined as follow:

\[ H_{in} \times B_{out} = 3 \times 3 = 9 \]
\[ B_{in} \times D_{out} = 1 \times 1 = 1 \]
\[ B_{in} \times C_{out} = 1 \times 2 = 2 \]
\[ C_{in} \times H_{out} = 2 \times 2 = 4 \]
\[ C_{in} \times E_{out} = 2 \times 1 = 2 \]
\[ E_{in} \times F_{out} = 2 \times 2 = 4 \]
\[ F_{in} \times D_{out} = 1 \times 1 = 1 \]

---

\(^6\) The number of specific stubs generated by choosing \(E(H, B)\) being deleted first is the same as choosing \(E(A, C)\).
Again, we do not further detail the application of the algorithm on the example. The remaining steps involve breaking associations (H, B), (E, F), and (C, H) in that order, thus leading to Figure 8 which shows all the SCCs which have been identified during the process and the corresponding broken associations, as well as the final test order. Four classes (B, C, F, H) need to be stubbed and four associations are broken, leading to four specific stubs. We therefore obtain a number of specific stubs that is comparable or better, depending on the traversal selected, to the results obtained with Le Traon et al (see Table 1). This was one of our two main objectives, along with not breaking aggregation or inheritance dependencies.

![Figure 8 Applying the new approach to Figure 2](image)

A is tested using stub(C, A);
E is tested using A and stub(F, E);
C is tested using A, E, and stub(H, C);
H is tested using C and stub(B, H);
D is tested using A and H;
F is tested using E and D;
B is tested using C, D and H;
G is tested using F and B.

3.4 Discussion

Table 1 summarizes the results we obtained (from the example in Figure 2) with all three strategies to test ordering with cycles. The most salient results are that, as expected because of the weight definition they use, Le Traon et al shows a consistently lower number of classes stubbed (3 versus 4 and 5, for our strategy and Tai and Daniels, respectively). Because of the way we define weights, which specifically aims at minimizing the number of broken associations, we obtain a number of specific stubs (4) that is consistently lower than le Traon et
(4, 6, and 7) and Tai and Daniels (5). Because the latter technique is not optimal, either in terms of classes to be stubbed or specific stubs, it should probably not be used. Problems are encountered when associations are cut when not participating to cycles, just because they cross major levels. This explains, in our example, why 5 associations are deleted versus four with our new strategy. It’s main advantage lies in its simplicity but the other two algorithms can easily be supported by tools.

As described in previous sections, the two other approaches have different objectives, and we have argued that ours considers a better measure of integration cost: Ours minimizes the number of specific stubs, whereas Le Traon et al’s strategy minimizes the number of classes stubbed. However, the two approaches are both based on a recursive identification of SCCs using the same algorithm, which is linear in the number of classes in the ORD [22]. Since this identification of SCCs is by far the most expensive part in both algorithms, they have the same overall time complexity.

In addition, Le Traon et al may lead to the unacceptable cases where aggregations or inheritance dependencies are broken, thus leading to an even higher integration cost. Last, a weight characterizing associations is much more flexible and allows for numerous extensions such as forbidding the deletion of associations leading to control, state-dependent classes (that are hard to stub) or any other complexity measurement. These last points are, however, the subject of further research.

Another issue is that both Le Traon et al and our strategy do not specify what to do when two or more classes/dependencies have the same weight. We have seen in the example above (see also Table 1) that, when using Le Traon et al’s strategy, choosing one successor class to traverse over the other would not only change the integration order but also the resulting number of specific
stubs. For example, when the initial class is H in our example, we can then choose among three classes to traverse of equal weight (A, E, F). Depending on which one we choose we obtain either 6 or 7 specific stubs. In the case of our strategy, the integration order changes but the number of specific stubs remains the same as one association is broken in all cases. In the next section, the above observations will be confirmed by experimental results on 5 application systems.

<table>
<thead>
<tr>
<th>Classes stubbed</th>
<th>Tai and Daniels</th>
<th>Le Traon et al</th>
<th>New strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traverse starting with H</td>
<td>Traverse starting with H</td>
<td>Traverse starting with G</td>
<td></td>
</tr>
<tr>
<td>Choose vertex A</td>
<td>Choose vertex E</td>
<td>Choose vertex F</td>
<td></td>
</tr>
<tr>
<td>H, A, F</td>
<td>H, E, A</td>
<td>H, F, A</td>
<td></td>
</tr>
<tr>
<td>H, F, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, C, F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3H, 3A, 1F</td>
<td>3H, 2E, 1A</td>
<td>3H, 1F, 2A</td>
<td>1B, 2C, 1F</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1 Summary of Results

4 CASE STUDIES

In this section, we first introduce the five case studies we have selected, and then provide some details on how we reverse-engineered the ORDs from the corresponding Java source code. Then, we present the results of our comparison of the three graph-based ordering techniques on the five selected application systems. Note that this comparison does not only look at the number of stubs required when applying each strategy but also provides the stubbing complexity of the produced test order in terms of methods and attributes that may potentially be involved in the stubs (the measures are described in Section 4.2).

4.1 Description of the case studies

The first system is an Automated Teller Machine (ATM) simulation (the classes connected to hardware devices are missing). The class diagram is made of 21 classes and 67 relationships,
and contains 30 cycles\(^7\) involving 8 of the 21 classes. The second system, named Ant, is part of the Jakarta project (http://jakarta.apache.org). Ant creates and maintains open source solutions on the Java platform for distribution to the public at no charge. The Ant system is a Java based build tool similar to the make tool on Unix platforms: It maintains, updates, and regenerates related programs and files according to their dependencies (e.g., compilation units). The class diagram consists in 25 classes and 83 relationships, and contains 654 cycles involving 12 of the 25 classes. The third example, named SPM (Security Patrol Monitoring), is a course project implemented by a graduate student at Carleton University. This system monitors security zones (e.g., authorized entry/exit) and patrols (e.g., schedules). The class diagram consists in 19 classes and 72 relationships, and contains 1,178 cycles involving 15 out of the 19 classes. The fourth example, BCEL (Byte Code Engineering Library), also comes from a subproject of Jakarta Project, and is intended to give users a convenient tool to analyze, create, and manipulate binary Java class files. We used the \texttt{org.apache.bcel.classfile} package of version 5.0 as our example (http://jakarta.apache.org/bcel/index.html). The class diagram is made of 45 classes, and 294 relationships, and contains 416,091 cycles involving 41 out of 45 classes. The last application system, named dnsjava or simply DNS in this article, is an implementation of Domain Naming System in Java: i.e., it provides network naming services (http://www.xbill.org/dnsjava/). The DNS class diagram consists in 61 classes and 276 relationships, and contains 16 cycles involving 10 out of 61 classes.

\(^7\) We count here the number of \textit{elementary} circuits (algorithm provided in [23]), i.e., each class appears once and only once in each circuit.
These five application systems\(^8\) were chosen because they were deemed to be of sufficient size and of varying complexity, so as to assess the effectiveness of the three graph-based approaches. ATM, Ant, SPM, and BCEL have class diagrams of reasonable (and comparable) sizes (between 19 and 45), but with very different numbers of cycles (from 30 for ATM to 416,091 for BCEL). On the other hand, the DNS system has the most important number of classes (and almost the same number of relationships as BCEL), but the smallest number of cycles (fewer number than ATM, Ant, and SPM)! This shows the topography of class diagrams can vary a great deal across application systems. Further details about the systems are provided in Table 2.

<table>
<thead>
<tr>
<th>System</th>
<th>Classes</th>
<th>Usages</th>
<th>As and Ag</th>
<th>Compositions</th>
<th>Inheritance</th>
<th>Cycles</th>
<th># Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>21</td>
<td>39</td>
<td>9</td>
<td>15</td>
<td>4</td>
<td>30</td>
<td>1390</td>
</tr>
<tr>
<td>Ant</td>
<td>25</td>
<td>54</td>
<td>16</td>
<td>2</td>
<td>11</td>
<td>654</td>
<td>4093</td>
</tr>
<tr>
<td>SPM</td>
<td>19</td>
<td>24</td>
<td>34</td>
<td>10</td>
<td>4</td>
<td>1178</td>
<td>1198</td>
</tr>
<tr>
<td>BCEL</td>
<td>45</td>
<td>18</td>
<td>226</td>
<td>4</td>
<td>46</td>
<td>416,091</td>
<td>3033</td>
</tr>
<tr>
<td>DNS</td>
<td>61</td>
<td>211</td>
<td>23</td>
<td>12</td>
<td>30</td>
<td>16</td>
<td>6710</td>
</tr>
</tbody>
</table>

**Table 2 Detailed information for the 5 case studies**

Though the number of classes involved in the 5 selected systems may seem modest by comparison with some of the systems that are commonly developed across the software industry, recall that in large systems the strategies we describe in this article would be used at different levels of integration, in a stepwise manner. For example, graph-based algorithms would first be used to integrate classes into lower-level subsystems and then lower-level subsystems into higher-level subsystems, step by step until the system is entirely integrated. It is then unlikely that a given subsystem contains more than a couple hundred classes or more than a few dozens lower-level subsystems. Note that in the case where lower-level subsystems are integrated, the

---

\(^8\) We made a conscious effort not to use libraries but application systems, in order to use case studies representative of the type of systems on which the techniques would typically be used and so as to avoid the peculiar class diagram topologies encountered in libraries (e.g., in [14], 4 of the 6 systems used are libraries and, for example, the Java Library shows 8000 cycles that are broken using 7 stubs).
graph-based techniques presented here are also required and used in the same manner, based on
an ORD which nodes are subsystems and which dependencies are reflecting the dependencies of
the classes they contain, focusing exclusively on those that cross subsystem boundaries.

4.2 Estimating the cost of a test order

As for the cost of test orders, since this cannot be measured directly, even in the context of an
industrial field study, we decided to estimate the orders’ stubbing complexity and considered two
simple measures of the coupling involved in class relationships. More measures could have been
used [3], but those two measures are enough to capture the two main types of coupling involved
(method and attribute coupling) while avoiding complex reverse-engineering or analyses of the
UML diagrams:

1. **Attribute coupling**: The number of attributes *locally* declared\(^9\) in the target class when
references/pointers to instances of the target class appear in the argument list of some
methods in the source class, as the type of their return value, in the list of attributes (data
members) of the source class, or as local variables of methods. This complexity measure
counts the (maximum) number of attributes that would have to be handled in the stub if
the dependency were broken. In the case of inheritance, we count the number of attributes
declared in the parent class.

2. **Method coupling**: The number of methods (including constructors) *locally* declared\(^9\) in
the target class, which are invoked by the source class methods (including constructors).
This complexity measure counts the number of methods that would have to be emulated
in the stub if the dependency were broken. In the case of inheritance, we count the

\(^9\) We do not count inherited attributes (and methods) as this would lead to counting them several times when
measuring the stubbing complexity of an order.
number of methods declared in the parent class. Note that this is an approximation as some of the methods can be overridden.

4.3 Reverse-Engineering ORDs

Reverse-engineering an ORD means identifying classes and their relationships from source code. The identification of inheritance and implementation (in the case of Java) relationships is straightforward, as there exist specific constructs in object-oriented languages for these two relationships. The work is more complicated when it comes to the other relationships we are interested in, i.e., usages, associations, aggregations, and compositions, which are mapped to As and Ag in the ORD (see the mapping in [5]). Indeed a structural analysis of the source code (e.g., .java files) can only indicate the presence of an association or usage, but determining that an association is in fact an aggregation/composition requires more understanding of the semantics of the system. We identify associations and usages as follows:

- When class A has an attribute of type class B then we define an association between A and B. This identification is further complicated by the presence of data structures provided by libraries (e.g., class LinkedList in Java). In that case, an analysis of the body of the methods is necessary in order to identify which types (classes) are stored in the data structure\(^{10}\). As a consequence, a manual analysis is required in order to identify possible compositions among these associations.

- When a method in class A has a parameter, a return value, or a local variable, the type of which is class B, then we define a usage dependency between A and B, provided that

\(^{10}\) This is not possible when one use the introspection mechanisms provided by Java, since the body of methods are not accessible. Note that in [14], the authors used the Java introspection mechanisms and do not provide detailed information regarding the reverse-engineering of the ORDs (e.g., how they identify usages), thus preventing us from comparing their results with ours.
there is no association already defined between A and B. Again, part of this activity requires access to the method bodies.

We have built a tool in order to automate, to the largest extent possible, the reverse-engineering of the ORDs. We used the Java Compiler Compiler (Java-CC), which is a Java Parser Generator (http://www.webgain.com/products/java_cc/). Java-CC reads a grammar specification (e.g., the Java grammar) and converts it to a Java program that can recognize matches to the grammar. It is also able to generate the syntax tree. We extended the classes used for building the syntax tree so that traversing the tree (produced by the parser generated by Java-CC) identifies the required information to build an ORD, and the coupling data. However, this analysis cannot be fully automated when semantic information is required in order to identify the correct type of relationship between two classes (e.g., identifying that an association is in fact a composition) and, in those cases, the tool indicates what parts of the source code should be analyzed by the user in order to complete the ORD reverse-engineering.

4.4 Results

Since each of the three strategies we compare in this article has some level of non-determinism (see Section 3.4), we ran the corresponding algorithms 100 times and performed random selections each time it was necessary (e.g., selecting an association when more than one association maximizes the weight function in an SCC). We thus generate distributions of numbers of stubs, as well as distributions of attributes and methods that may need to be stubbed. Each of the five case studies is presented in sequence below, and accompanied with a figure showing those distributions. In addition, statistical testing is performed to assess the statistical significance of the difference between the three investigated algorithms, for each of the stub, method, and attribute distributions. The Wilcoxon Rank-Sum non-parametric test [6] is used as
distributions are sometimes clearly not normal. This statistical test is based on comparing the value ranks of values across two independent samples and does not make any distribution assumption. We use a standard significance threshold of $\alpha = 0.01$ to draw our conclusions but do not report, due to space constraints, all the computed p-values of each individual test.

To come back on the issue of time complexity discussed earlier, note that, for each of the five case studies, executing 100 times each of the three algorithms took the order of a minute on a typical personal computer (500MHz, 128 MB). Thus the time complexity of these algorithms is unlikely to be an issue in practice, making them highly applicable in an industrial context where the impact of alternative designs on integration can be quickly evaluated.

Table 3 shows the results for the ATM system. First, in terms of number of stubs, we see that Le Traon et al’s technique produces a distribution of widely varying results across the 100 executions of the algorithm. This is not the case of Briand et al or Tai and Daniels’ approaches. This supports the conclusions regarding non-determinism we drew in Section 3.2.2. Furthermore, Briand et al clearly outperforms Le Traon and does slightly better than Tai and Daniels. The difference between the two former techniques are clearly practically significant as the difference in stubs has a significant probability of reaching 5 or more, which is quite large considering the number of classes in this system. In terms of attributes and methods, the differences between Briand et al and the two other techniques are even more acute. Attribute ranges are, for the three techniques: [39-67] (mean: 54), [67-240] (mean: 95), and [52-80] (mean: 67); As for methods, we get [13-19] (mean: 16), [13-75] (mean: 25), and 21. It is also noteworthy that some of the orders generated by Le Traon (4 among the 100 produced) break 4 inheritance relationships and 1 composition relationship. Statistical testing shows that, at the $\alpha = 0.01$ level of significance, all differences between Briand et al and the two other algorithms are statistically significant.
Practical significance is more subjective in nature, but we can clearly see that the differences with Le Traon and Tai and Daniels can be quite large, except for the latter one when considering the number of stubs.

<table>
<thead>
<tr>
<th>Briand et al.</th>
<th>Le Traon et al.</th>
<th>Tai and Daniels</th>
</tr>
</thead>
<tbody>
<tr>
<td># of stubs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>39 40 41 42 43 44 45 46 47 48</td>
<td>67 80 136 162 240</td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 2 4 6 8 10 12 14 16 18</td>
<td>0 2 4 6 8 10 12 14 16 18</td>
</tr>
<tr>
<td></td>
<td>13 19</td>
<td>13 19 39 64 75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 Results for the ATM system

As for the Ant system (see Table 4), the results confirm, in an even stronger manner (the system is more complex), the results obtained with ATM. Briand et al performs significantly better than Le Traon et al and Tai and Daniels in terms of stubs, attributes, and methods: All differences are significant at the $\alpha = 0.01$ level of significance. The magnitude of the differences can be quite large, thus clearly showing practical significance, e.g., the method means for the three techniques are 25, 78, and 89. Table 8 provides the means and medians for all three distributions. Regarding Le Traon et al, the data also shows that only 8 among the 100 test orders do not break inheritance or composition relationships: 9 of them break 1 inheritance and 1 composition; 61 break 2
inheritances and 1 composition; 13 break 3 inheritances and 1 composition; 2 break 1 inheritance and no composition; 7 break 2 inheritances and no composition.

<table>
<thead>
<tr>
<th></th>
<th>Briand et al.</th>
<th>Le Traon et al.</th>
<th>Tai and Daniels</th>
</tr>
</thead>
<tbody>
<tr>
<td># of stubs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>152 157 178</td>
<td>151 239 276 295</td>
<td>257 328 335</td>
</tr>
<tr>
<td></td>
<td></td>
<td>302 310 315 327</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>335 341</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19 26 30</td>
<td>25 44 64 67 72</td>
<td>88 90 92 94 99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>74 88</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>90 92 94 99</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 Results for the Ant system

Results for the SPM system confirms the results of the two previous as they are also clearly in favor for Briand et al.’s strategy (Table 5): the number of stubs and the corresponding attribute and method costs are constantly lower than those produced by Le Traon et al and Tai and Daniels. The differences are all statistically significant at the $\alpha = 0.01$ level of significance. In many cases, this is clearly visible just by looking at the distributions where the values ranges are not even overlapping. For example, the method average of the three techniques (Table 8) are 27, 64, and 39. In addition, it is interesting to note that the 100 orders produced by Le Traon et al’s strategy consistently break 2 compositions.
Results for the SPM system are reported in Table 5. Briand et al. and Le Traon et al.’s strategies are consistently better than Tai and Daniels ($\alpha = 0.01$). However, in terms of stubs and attributes, Le Traon et al performs better than Briand et al, though the difference in stubs is very small and not practically significant. The number of methods though is clearly much smaller (by a magnitude of about 4 times) for Briand et al. So, overall, results are mixed based on these three criteria. However, though the numbers of stubs are similar, the 100 orders produced by Le Traon et al’s strategy consistently break 22 inheritance relationships, thus in the end favoring the results of Briand et al.
Table 6 Results for the BCEL system

Regarding the DNS system, results are once again clearly in favor for Briand et al.’s strategy (Table 7). The difference between Briand et al and Le Traon et al’s strategies are statistically ($\alpha = 0.01$) and practically significant. For example, Table 8 shows method averages of 11, 77, and 32, for Briand et al, Le Traon et al, and Tai and Daniels. Furthermore, the 100 orders produced by Le Traon et al’s strategy consistently break a minimum of 2 inheritance relationships.
The results obtained by the three graph-based techniques are summarized in Table 8. For each of the systems, we indicate the mean and median values of both attribute and method costs. Highlighted cells show the minima, and are all (except in one case discussed above) produced by Briand et al strategy. Overall, from the discussions above and Table 8, we can draw a number of conclusions regarding the results of our case study:

- Briand et al clearly outperforms, both in the statistical and practical sense, the two other graph-based techniques to minimize stubbing effort. It does so with respect to the three criteria we have measured: stubs, attributes, and methods.

- Le Traon et al performs, in general, better than Tai and Daniels but that is not consistently the case as it depends on which system and criterion we look at.

- Le Traon et al generates, in all cases, a distribution of stubs, thus showing the non-determinism of the algorithm in practice.
- Le Traon et al. tends to break a significant number of inheritance and/or composition relationships.

<table>
<thead>
<tr>
<th></th>
<th>Briand et al.</th>
<th>Le Traon et al.</th>
<th>Tai and Daniels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Attribute</td>
<td>Method</td>
<td>Attribute</td>
</tr>
<tr>
<td>ATM</td>
<td>54.16</td>
<td>16</td>
<td>94.87</td>
</tr>
<tr>
<td></td>
<td>54</td>
<td>16</td>
<td>87</td>
</tr>
<tr>
<td>Ant</td>
<td>163.38</td>
<td>24.62</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>157</td>
<td>26</td>
<td>314</td>
</tr>
<tr>
<td>SPM</td>
<td>148.36</td>
<td>26.87</td>
<td>265.94</td>
</tr>
<tr>
<td></td>
<td>148</td>
<td>27</td>
<td>254</td>
</tr>
<tr>
<td>BCEL</td>
<td>117.12</td>
<td>72</td>
<td>92.38</td>
</tr>
<tr>
<td></td>
<td>118</td>
<td>72</td>
<td>92</td>
</tr>
<tr>
<td>DNS</td>
<td>23.05</td>
<td>11</td>
<td>74.5</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>11</td>
<td>84</td>
</tr>
</tbody>
</table>

Table 8 Attribute and method costs for orders produced by the three strategies (mean, median)

5 CONCLUSIONS

We have performed a systematic analytical and empirical evaluation of existing graph-based techniques to generate test orders when integrating classes. We showed that in cases where associations cross major levels and are not involved in cycles, the strategy by Tai and Daniels [21] leads to the unnecessary use of stubs. Regarding the strategy by Le Traon et al [15], though it clearly optimizes the number of classes to be stubbed, it can lead in practice to the deletion of aggregation or inheritance dependencies, and the output of their algorithm depends on some arbitrary search choices that may produce very different results. Those statements are demonstrated in analytical terms but also clearly supported by our experimentation on five application systems of non-trivial and varying complexity.

More importantly, we have shown that the third graph-based strategy we propose, which integrates the fundamental principles of Le Traon et al and the dependency weighting principles of Tai and Daniels, does not have the problems mentioned above, and performs better in terms of
stubs, attributes, or methods those stubs may need to emulate. Differences have shown to be statistically (at the $\alpha = 0.01$ level of significance) and practically significant in the 5 reported case studies.

Future work will include the development of more precise measures to better estimate the cost of subbing and the graph-based search for optimal orders in the context of constraints (e.g., one may want to big-bang test classes in a strongly connected component instead of break the cycles it contains; one may want to consider that some parts of the class diagram are not available because not already implemented or unit tested). This will probably require the use of optimization techniques, as it has already been explored recently with genetic algorithms [4], and we believe this is a new direction of research that is worth further investigating and involves entirely new challenges.

ACKNOWLEDGEMENTS

This work was partly supported by the NSERC-CSER consortium and Mitel Networks through a CRD grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is part of a larger project on testing object-oriented systems with the UML (TOTEM: http://www.sce.carleton.ca/Squall/Totem/Totem.html). We would like to thank Jie Feng and Nian Nian Ding for their help in automating the experiment and collecting the data. The authors would like to thank the associate editor and anonymous reviewers for their helpful and constructive comments.

REFERENCES


