A Genetic Algorithm for Computing Class Integration Test Orders for Aspect-Oriented Systems

Romain Delamare and Nicholas A. Kraft
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487-0290
{rdelamare,nkraft}@cs.ua.edu

***Abstract*—In this paper we present an approach for the class integration test order problem in aspect-oriented programs. Several approaches have been proposed for aspect-oriented systems, but the proposed approach is the first, to our best knowledge, to consider the indirect impact of aspects. This approach relies on a genetic algorithm and can reduce the testing efforts when many methods are indirectly impacted by aspects. We detail the algorithm and then discuss its parameters. The approach has been implemented for AspectJ systems, and to validate it, has been applied to a motivating example.**

***Keywords*-software testing; class integration test order; aspect-oriented programming**

I. Introduction

Integration testing is the step of the testing process where classes are composed and tested together to find faults related to interactions between classes [1]. To facilitate fault localization and to minimize fault interactions, a class should be tested after the classes on which it depends. If there are cyclic dependencies between classes, which is common in large software systems [2], a *stub* is needed for every untested class on which the class under test depends. A stub acts as a surrogate for the actual class by mimicking its interface and behavior. Once the actual class has been tested, the stub is no longer needed. Creating a stub is costly, especially in the case of complex classes or relations, and techniques for computing a class integration test order aim to find an order that minimizes the stubbing effort.

In the case of aspect-oriented programming [3], aspects are tightly coupled with the classes in which they are woven. As an aspect must be woven to be executed, testing the aspect first means creating a stub environment into which it can be woven. This is a difficult and complex task, and it is thus more reasonable to test the aspect after the classes where it is woven. This means that the test suites for these classes must be modified after the aspect has been added.

In most approaches for the integration testing of aspect-oriented systems, the base program is tested first without the aspects, and then the woven program (which includes the aspects) is tested. This approach is simple and permits reuse of class integration test order techniques for object-oriented systems. However, it is not optimal when many classes are indirectly impacted by aspects. A class is indirectly impacted by an aspect if it relies on classes impacted by this aspect. When an indirectly impacted class is tested before the aspect, it is first tested with a test suite that does not account for the aspect, then, after the aspect has been tested, it is tested with a test suite that accounts for the modifications introduced by the aspect. This implies an extra testing effort that could be avoided using a fine grained approach that produces an integration order in which classes and aspects are interspersed.

Class integration test order techniques for object-oriented systems cannot be reused on aspect-oriented systems without being adapted. Aspects add a new kind of relationship between the aspects and the classes where they are woven, and they can introduce new attributes or methods to existing classes. The impact of the aspects also needs to be measured and taken into account to produce an optimal integration order. Finally, class integration test order techniques for object-oriented systems are usually graph-based, and cannot be adapted to account for the problems introduced by the aspects, because these existing techniques consider only binary relationships between classes.

In this paper we propose a genetic algorithm which produces a class integration test orders for aspect-oriented systems. Specifically, the algorithm produces integration test orders in which aspects are tested after the classes they directly impact but before classes they indirectly impact. Genetic algorithms often perform well for optimization problems (such as the class integration test order). Further, a genetic algorithm can use global information, so the result can take into account all the new implications induced by the aspects.

The contribution of this paper is a genetic algorithm based approach to determining a class integration test order in the context of aspect-oriented systems. Specifically, the *fitness function* is responsible for selecting the best solution, and is thus the key element of the approach. The proposed approach has been validated using a medium-sized AspectJ program.

In Section II, we introduce the class integration test

order problem in the context of aspect-oriented systems, including a motivating example. In Section III, we present the proposed approach. In Section IV, we describe our implementation. In Section V, we discuss the parameters of the genetic algorithm. In Section VI, we present the related work. Finally, Section VII concludes the paper.

II. Integration test order in aspect-oriented programs

In an object-oriented system, if a class `C1` depends on (i.e., uses) a class `C2`, then `C2` should be tested before `C1`. However, if `C2` also depends on `C1`, the hardest class to stub (which depends on the types of relations, the complexity of the class, etc.) should be tested first.

In an aspect-oriented system, if an aspect is woven within a class, they are strongly connected. The aspect needs to be woven to be tested as it cannot be executed alone. The behavior of the class is not complete without the aspect.

It is best to test an aspect after the classes where it is woven. Testing the aspect first means creating a "stub" environment where the aspect can be woven, which is not trivial and could be as complex as the stubbed class. Testing the class first means writing a partial test suites that will be modified after the aspect has been woven, which is less complex even though it still requires extra testing effort.

From this conclusion, the most used approach for the integration testing of aspect-oriented programs, which we will call the *incremental* approach, is to test the base program first and then to test the program with its aspects woven. Zhou *et al.* [4] or Ceccato *et al.*[5] advocate the use of such an approach, adding a few aspects at a time, to improve fault localization. Xu *et al.* [6] present an approach for generating test cases for integration testing from state models. Their approach is based on an incremental process where state models for the base program are first used to generate test cases. These state models are then modified to take aspects into account, and new test cases are finally generated.

The incremental approach has several advantages. First it is easier to implement; testing the base program is done using classic techniques and aspect-oriented specific techniques need only to focus on the interactions between the aspect and the code. It may also improve fault localization, although aspects can have such significant impacts on the base program that it is difficult to have a precise fault localization [7], [8].

The main problem of the incremental approach is that it may require a lot of unnecessary testing effort if there are many classes that are impacted indirectly (but not directly) by an aspect. A class is *directly impacted* if an aspect is woven within it. A class is *indirectly impacted* if no aspect is woven within it, but it depends on impacted classes (i.e., classes directly or indirectly impacted). The indirect impact of an aspect may require a class to be retested, or test cases to be partially rewritten. Indeed, the indirect impact of the aspect can make obsolete existing test cases or require new test cases. A proper class integration test order for aspect-oriented programs may reduce this testing effort. In this section we present an example to illustrate this problem and to motivate our approach.

A. Bank Example

The Bank example [9] is a system that manages bank accounts. Three kinds of user can access the system: clients, employees, and administrators. Clients can access their accounts to check the balance or to make deposits and withdrawals. Employees can create different kind of accounts for clients and manage them. Administrators have access to maintenance functions as well as the creation of employee accounts.

Figure 1 shows an excerpt of the UML class diagram of the Bank example. The system is implemented in Java, and has 28 classes, 3 interfaces, and 101 methods.

The system is divided in four parts: model, view, controller, and security. The first three parts implement the Model-View-Controller design pattern [10]. The model package encapsulates the core concerns of the application. The view package offers a graphical user interface to interact with the system. The controller package handles the input of the user from the views and interact with the model. The controller package uses the command design pattern [11]. The security package (not displayed on the class diagram) allows users to authenticate and manages access control.

The access control concern is implemented with an AspectJ aspect, which is displayed in Listing 1. There are three different annotations that are used to control the access on methods. A method of the class `Account` (or any of its subclasses) annotated with `@OwnerOnly` can only be executed if the current user is one of the owners of the account or an employee. A method annotated with `@EmployeesOnly` (respectively `@AdminsOnly`) can only be executed if the current user is an employee (respectively an administrator).

The `Security` aspect has three advices. The first one (lines 6–11) manages the "owner only" access control and is only woven in the `Account` class and its subclasses. The second one (line 13–18) manages the "employees only" access control and is only woven in the `Bank` class. Finally, the last one (line 20–25) manages the "administrators only" access control and is woven in the `Bank` class and in the `User` class.

B. Integration Testing of the Bank Example

The `Security` aspect is only woven in the model package, and directly impacts only four classes. The aspect relies on those classes, but as the *obliviousness*[1] property

[1]The obliviousness property [12] is satisfied if the base program has no knowledge of the aspect that are woven within it (i.e., the base classes do not reference the aspect).

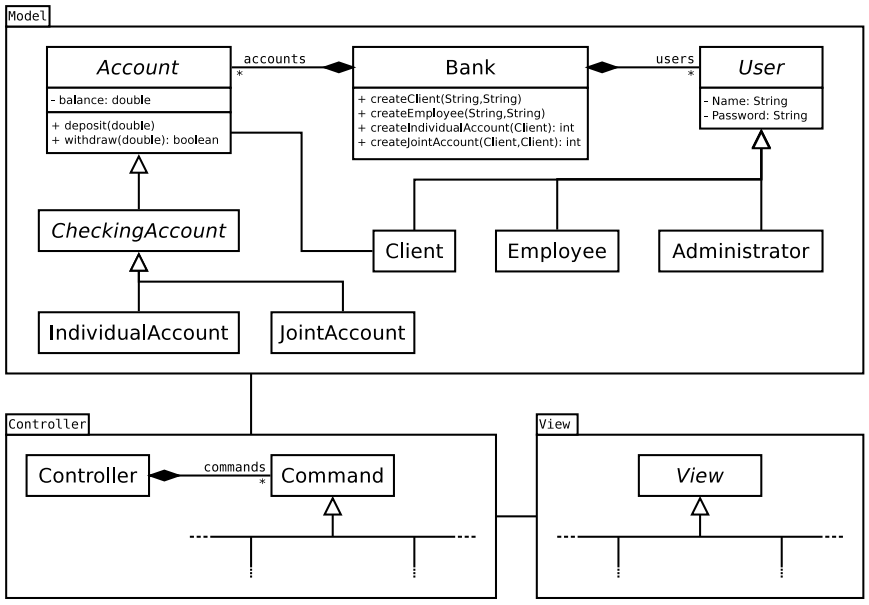

Figure 1. Excerpt of the UML class diagram of the Bank example

```
1  public aspect Security {
2      pointcut ownerOnly(): execution(@OwnerOnly
           * Account+.*(..));
3      pointcut employeesOnly(): execution(
           @EmployeesOnly * *.*(..));
4      pointcut adminsOnly(): execution(
           @AdminsOnly * *.*(..));
5
6      before(): ownerOnly() {
7          User user = Login.instance.
               getCurrentUser();
8          Account account = (Account)
               thisJoinPoint.getTarget();
9          if(checkAccessOwner(user,account))
10             throw new AccessSecurityException(
                   user, account);
11     }
12
13     before(): employeesOnly() {
14         User user = Login.instance.
               getCurrentUser();
15         Object target = thisJoinPoint.getTarget
               ();
16         if(checkAccessEmployee(user,target))
17             throw new AccessSecurityException(
                   user, target);
18     }
19
20     before(): adminsOnly() {
21         User user = Login.instance.
               getCurrentUser();
22         Object target = thisJoinPoint.getTarget
               ();
23         if(checkAccessAdmin(user,target))
24             throw new AccessSecurityException(
                   user, target);
25     }
26 }
```

Listing 1. The Security aspect

is satisfied, the impacted classes do not rely on the aspect. Thus, during integration testing, the aspect should be tested after the four classes directly impacted.

But the aspect indirectly impacts the whole system. The controller accesses the model and the methods whose access is controlled. The views, in turn, access the controller. As the aspects impacts the control flow of the program, it is necessary to take into account its changes in the test suites written for the views and for the controller.

The incremental approach induces an extra testing effort. Testing the `Security` aspect after all the classes of the system means that the test suites for the classes that are indirectly impacted must take into account this impact at the end of the integration testing. In the best case scenario, test cases must be added, but it is possible that some test cases must be modified or others removed, as they are obsolete. This implies more testing effort.

This testing effort could be partially reduced by using a *combined* approach, where the produced integration test order mixes classes and aspects. For instance, the controller and the views could be tested after the `Security` aspect. As the aspect does not depend on the controller and the views, no stubbing and thus no extra effort are required. When the controller and the views are tested, the test suites will directly take into account the impact of the `Security` aspect.

In other examples, the extra testing effort implied by testing the aspects at the end can be much more important than in this example. In the Bank example, only the control flow is affected by the aspect. Aspects affecting the data flow or modifying the control flow more drastically will most likely affect a large number of test cases.

Class integration test order techniques for object-oriented systems cannot directly be applied on aspect-oriented systems and need to be adapted. Aspects can be considered as classes but their weaving introduces a new kind of relationship, and inter-type definitions – attributes or classes introduced by an aspect – also add complexity. Techniques for object-oriented systems do not take into account the direct and indirect impact of the aspects, which, as we have discussed in this section, is important to minimize the testing effort.

A fine-grained approach, that specifically targets aspect-oriented programs, could lead to more efficient integration testing. This approach must be able to identify classes that are indirectly impacted by an aspect and that can be tested after this aspect. In the following section we present such an approach, which uses a genetic algorithm.

III. Approach

The goal of the proposed approach is, for an aspect-oriented system, to find a class integration test order in which aspects are tested after the classes they directly impact, and before the classes they indirectly impact. As discussed in Section II, such an order should reduce the testing effort by avoiding the modification of test suites for indirectly impacted classes.

Most solutions for the class integration test order in object-oriented systems are graph-based [13], [14], [2], [15]. They rely on an object relation diagram (ORD)[2], which is a directed multigraph in which the nodes represent classes and the edges represent relationships between classes. The goal of these solutions is to reduce the stubbing effort. To minimize stubbing effort in the presence of cyclic dependencies, a feedback arc set must be computed for the ORD. Because this problem is NP-complete, graph-based approaches rely on heuristics in which edges are assigned weights based on the perceived effort to create the corresponding stub. Strongly connected components are computed and then eliminated by iteratively removing edges with the smallest weights.

As first discussed by Briand *et al.* [16], graph-based solutions may not be satisfactory. These solutions try to find an optimal solution at each strongly connected components, so each decision is based on local, rather than global, information.

In the case of aspect-oriented programs it is critical to reason on global information. As discussed in Section II, the indirect impact of the aspects needs to be taken into account for determining a class integration test order. This is not possible by considering sub-parts of the system, or by evaluating single relations. Removing an edge representing the weaving in an aspect may be the best solution for a local strongly connected component, but may also require a lot of extra testing work in other parts of the system because of the indirect impact of the aspect. Thus, global information is needed to produce a class integration test order in the context of aspect-oriented programming.

Genetic algorithms, by allowing the evaluation of a whole order, can solve this problem. In a genetic algorithm, each candidate solution is evaluated using a fitness function. Thus the whole system can be used to evaluate a solution. Genetic algorithms are often used for *global optimization* problems [17], [18], [19], [20].

The solution we propose is based on the genetic algorithm which we describe in this section.

A. Genetic Algorithm

Genetic algorithms are evolutionary algorithms that mimic natural evolution. A population of candidate solutions is created then evolved. At each generation the best solutions are selected and crossed-over. During a cross-over, parts of two solutions are mixed to produce a new solution. Hence, at each new generation, new solutions are produced.

A genetic algorithm requires a *genetic representation* and a *fitness function*. The genetic representation is the way in which solutions are encoded in *chromosomes*. The fitness function evaluates each solution by assigning it a *fitness value*. The higher the fitness value, the better the solution.

The algorithm is initialized with a random population. The population size may vary depending on the problem. The starting population covers the entire range of possible solutions.

A new population is generated with selection and reproduction. In our approach, chromosomes are selected using a *roulette wheel* selection [21], which proportionally gives chromosomes with higher fitness values a better chance to be selected. Each time a chromosome is selected there is a chance that a *crossover* happens (depending on the *crossover rate*). During a crossover, two chromosomes have sub-parts swapped between each other, creating a new chromosome. There is also a small chance that a chromosome is *mutated* (depending on the *mutation rate*). During mutation the chromosome is randomly modified. During reproduction, *elitism* can be used to avoid losing the best solutions. In such a case, the chromosomes with the best fitness value are automatically selected for the new population.

The genetic algorithm stops when the the termination condition has been reached. For example, the algorithm may stop if the optimal solution has been found (the maximal fitness value has been achieved), if no better solution is produced, or if a certain number of generations has been reached. In our problem, determining the maximal fitness value is as hard as determining the optimal solution, except in the trivial case where there is no cycle in the graph. Thus, we have chosen to stop the algorithm after a constant number of generations.

The parameters of the genetic algorithm are thus:

[2]Which is also, and more accurately, called a class dependency diagram

- the population size
- the crossover rate
- the mutation rate
- the maximal number of generations

These parameters are discussed in Section V.

B. Genetic Representation

In the case of class integration test order, it is best to opt for a *permutation encoding*. In a permutation encoding, each chromosome has the same number of *alleles*. Each alleles represents a class, and the rank of this class is the rank of the allele. So, the first class is the class represented by the first allele, an so on. Each class is assigned an arbitrary number. Chromosomes are thus permutations of the list $(1, \ldots, n)$ where n is the number of classes in the system.

In this work we have chosen a single point crossover operation, which is simpler to implement in the case of a permutation encoding. A random point is selected: the beginning of the first chromosome is copied up to this crossover point, the rest is composed by taking the remaining integers in the same order as in the second chromosome. For instance if the chromosomes $(6, 7, 1, 3, 4, 2, 5)$ and $(2, 1, 7, 5, 4, 6, 3)$ are crossed-over, and the random crossover point is 3, the resulting chromosome is $(6, 7, 1, 2, 5, 4, 3)$.

Depending on the mutation rate, each allele can be mutated. When an allele is mutated, it is permuted with another allele, randomly chosen. Several mutations can occur in a single chromosome, though as the mutation rate is low it is unlikely.

C. Fitness Function

The fitness function must take into account the relations between classes and the impacted methods. To reduce the stubbing effort it is necessary to reduce the number of stubbed relations. As discussed in Section II, aspects can impact methods and require an extra testing effort. Listing 2 shows the algorithm of the fitness function (in pseudo-code).

In the proposed approach, there are seven types of relations: *inheritance*, *association*, *composition*, *dependency*, *polymorphic*, *owned element*, and *weaving*. The first six relations were used in previous work [14], [2]. An inheritance relation is a relation between a class and one of its superclasses. An association relation is a relation between a class and the type of one of its attributes. A composition is an association where the object referenced by an attribute's life cycle depends on the source's life cycle. A dependency relation is a relation between a class and one of the classes it uses (for instance as a parameter of a method). An owned element relation is a relation between a class and one of the classes defined inside it. Finally, a weaving relation is a relation between an aspect and one of the classes where one of its advices is woven.

The value of each relation is added to the fitness value if the source of the relation is tested after the target (lines 13–14). The goal is to give a better fitness value to solutions that have an order that requires less stubbing. For each type of relation, we associate a value which represents the difficulty to stub the relation. The higher the value, the more difficult it is to stub this type of relation.

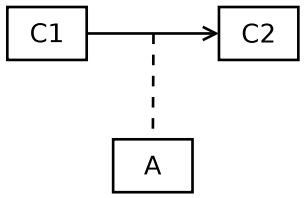

Figure 2. An object relation diagram illustrating a relation introduced by an aspect via an inter-type definition

Table I
THE THREE DIFFERENT TYPES OF ORDER FOR AN INTER-TYPE DEFINITION (CLASS NAMES FROM FIGURE 2)

Case	Orders	Value added to fitness
1	{C2,A,C1} {A,C2,C1}	relation value
2	{C1,C2,A} {C2,C1,A}	(relation value)/2
3	{A,C1,C2} {C1,A,C2}	0

The case of inter-type definitions is more complicated. Inter-type definitions are methods or attributes that are added by an aspect. It means that a relation between two classes can be added in the system by an aspect. In that case there are six possible orders that we classify into three different cases. Figures 2 depicts an object relation diagram (ORD) where a class `C1` has a dependency towards `C2` because of an aspect `A`. The dashed line between the relation and the aspect symbolizes the fact that the relation is introduced by the aspect. The three cases are detailed, with the associated value, on Table I. The best case (case 1) occurs if both `A` and `C2` are tested before `C1`. In that case, when `C1` is tested, no stub is required (because both `A` and `C2` have already been tested). If both `C1` and `C2` are tested before `A` (case 2), `C2` is still tested before the relation is added, but it is necessary to go back and retest `C2` to take the relation into account. Finally, if `A` and `C1` are tested before `C2` (case 3), it is necessary to stub `C2`, which was not required in the two other cases. In case 1, the value of the relation is added to the fitness value because it is the best case (lines 7–8). In case 2, there is no stubbing, but the order is less optimal, so only half of the value of the relation is added to fitness value (lines 9–11). Finally in case 3, the worst case, nothing is added to fitness value.

For each aspect, the number of impacted methods that

```
1  fitness <= 0
2  previous_types <= []
3  impacted_types // maps aspects to a list of (type,impacted_methods_nb) pairs
4  for t in chromosome:
5      for r in t.outgoing_relations:
6          if r.is_inter-type_definition:
7              if r.declaring_aspect in previous_types and r.target in previous_types:
8                  fitness <= fitness + r.type.value
9              else if chromosome.getIndex(t) < chromosome.getIndex(r.declaring_aspect) and
10                     chromosome.getIndex(r.target) < chromosome.getIndex(r.declaring_aspect):
11                 fitness <= fitness + r.type.value/2
12         else:
13             if r.target in previous_types:
14                 fitness <= fitness + r.type.value
15     for p in impacted_methods[t]:
16         if not p.type in previous_types:
17             fitness <= fitness + p.impacted_methods_nb
18     previous_types <- t
```

Listing 2. Algorithm of the fitness function

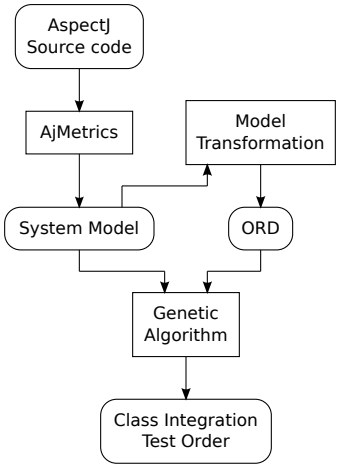

Figure 3. Overview of the implementation

are tested after the aspect is added to the fitness value (lines 14–16). As discussed in Section II, if a method is impacted by an aspect and if this aspect is tested after the class containing this method, it may be necessary to modify or remove existing test cases, or to add new test cases. This increases the testing effort, so if a solution can avoid it, its fitness value is increased.

The values associated with each type of relation are parameters of the fitness function. These values can be adjusted and modified to reflect different the priorities given to each type of relation. In Section V we discuss these parameters of the algorithm.

IV. Implementation

To validate the approach and the selected parameters we have implemented the algorithm presented in Section III. This implementation, illustrated on Figure 3, is performed in three steps, which we describe in this section.

The first step consists in building a detailed model of the system. This step is done using AjMetrics[3], an Eclipse[4] plug-in initially developed for measuring metrics on AspectJ systems. This plug-in produces an EMF[5] (Eclipse Modeling Framework) model that can then be used to measure metrics. This model encapsulates all the classes and aspects, with their attributes, methods, as well as the invocations and weaving information. The plug-in works for AspectJ systems, but the produced model is language-independent, which makes the rest of the implementation reusable for other languages.

In the second step, the system model is transformed into object relation diagram (ORD), which represents the relations between the classes. The transformation from the system model to the ORD is straightforward: the system model is visited, and for each element that implies a relation (e.g., an attribute or an invocation), a relation is created.

In the third step, the genetic algorithm is actually executed, using both models. The system model is used for the encoding of the chromosomes and for computing the methods impacted by each aspect. The ORD is used to compute the fitness value of the chromosomes.

Computing the impacted methods: To compute the fitness value of a chromosome, as described in Section III-C, it is necessary to know the methods impacted by each aspect. This is done once at the initialization of the algorithm, using the system model.

For each advice in the system, the set of all the methods where it is woven is computed. Then the transitive closure is computed, to include all the methods that can reach the points where the advice is woven.

As the set of impacted methods is computed statically, it is necessary to perform an over-approximation. It is

[3]http://www.romain-delamare.net/development/AjMetrics.html
[4]http://www.eclipse.org
[5]http://www.eclipse.org/modeling/emf/

Table II
RESULT OF THE EXPERIMENTS ON POPULATION SIZE AND MAXIMUM NUMBER OF GENERATIONS, WITH OTHER PARAMETERS FIXED

Population size	Number of generations	Average fitness	Median fitness	Average time (s)
25	1.000	707.00	706.00	0.8
50	1.000	719.20	720.00	1.1
100	1.000	721.80	719.00	1.8
25	10.000	725.50	727.50	2.9
50	10.000	730.00	729.50	5.9
100	10.000	734.40	733.00	12.7
25	100.000	744.20	745.00	24.6
50	100.000	745.60	746.50	54.3
100	100.000	747.00	746.50	125.7

possible the method actually executed at runtime is a method overriding the method statically called. To reflect that, when we look for all the methods that can call a method m, we also look for all the methods that can call the methods overridden by m.

For a quick access during the evaluation of the fitness function, the numbers of impacted methods are stored in tables. For each aspect a there is a table that matches each class c of the system with the number of methods of c where at least one advice of a is woven.

V. DISCUSSION

The proposed genetic algorithm has several parameters that need to be properly set to obtain satisfactory results. These parameters have been described in Section III, and are the *population size*, the *crossover rate*, the *mutation rate*, the *number of generations*, and the *values for the relation types*.

The crossover rate and mutation rate were set using the literature. Grefenstette [22] has experimented with different parameters to optimize a genetic algorithm. Concerning the crossover rate, the results showed that a mutation rate higher than 0.05 was harmful, and the optimal mutation rate was 0.01. We thus used 0.01 for the mutation rate. We set the crossover rate at 0.80, as the best result were shown with a crossover rate between 0.60 and 0.88.

Population size and number of generations: The population size and the number of generations were determined empirically. The algorithm was run on the Bank example from Section II, with all other parameters fixed, with different combinations of the two parameters, covering the ranges of good values, according to the literature [22], [23].

Table II shows the results with different combinations of the population size and number of generations. The values used for population size were 25, 50, and 100. The values used for the number of generations were 1.000, 10.000, and 100.000. All nine combinations have been experimented. For each combination, the algorithm was run 10 times over the Bank example, and the results show the average and the median of the fitness values of the best solutions found. The average execution time (in seconds) is also shown.

The chosen parameters were 25 for population size and 100.000 for the number of generations. Increasing the number of generations has the most effect on the results. All the combinations using 100.000 generations give better results compared to the other combinations. Increasing the population size also improves the result, but in the case of 100.000 generations the improvement is very small, and the execution time drastically increases with the population size.

Values for the relation types: Except for the weaving relation type, the values associated with each type of relation have been set using previous work [2]. In the context of a graph-based class integration test order algorithm for object-oriented systems, several values for the relations were experimented, using the number of stubs as a metric. Values are represented as a vector, with the values of inheritance, association, composition, dependency, polymorphic and owned element, in this specific order. The tested vectors were (40,2,4,2,2,4), (20,2,4,2,2,4), (10,2,4,2,2,4), (5,2,4,2,2,4), and (2,2,20,5,20,20). The last one showed the best results, so we used its values for the approach presented in this paper. As this previous work was considering object-oriented systems, there were no weaving relation, and we thus need to determine it empirically.

The weaving value should be high enough to make sure that an aspect is tested after the classes where it is woven. If an aspect `A` is woven in five methods of a class `C`, the fitness value would be five if `A` is tested first (for the five impacted methods), and it would be the value of the weaving relation if `C` is tested first. As we want the class (which is directly impacted) to be tested first, the value of the weaving relation, in this example, should be at least five.

To determine the best value for the weaving relation, we executed the algorithm on the Bank example using different values, all other parameters being fixed. The tested values were 2, 5, 10 , 15, and 20. For each value the algorithm was run ten times. The expected result is that the `Security` aspect is ranked after the four classes it directly impacts, but before the 15 classes it indirectly impacts. With a value of 2 or 5, only a few solutions showed the expected results. Starting from a value of 10 or higher, the `Security` aspect was ranked approximatively at the same place, and always after the classes it directly impacts.

We recommend using a high value for the weaving relation, such as 20. These results show that, on this example, 10 is the minimum value for the weaving relation, but we cannot know for sure that 10 is a good value for other examples. As the experiment showed that a higher value does not impact the rank of the aspect, using a higher value is safer.

This experiment also validates the discussion of Section II. The proposed algorithm has been able to find a class integration test order in which the aspect is tested after the classes where it is woven, but is tested before the other classes. This finer grained approach allows the reduction of the testing effort on the classes that are indirectly impacted.

VI. Related Work

We begin this section by reviewing approaches to computing a class integration test order for object-oriented systems. As we have discussed previously, these approaches may be applicable to aspect-oriented systems, but they are unable to account for aspects and are thus unable to compute an optimal (or even near-optimal) integration test order. After reviewing such approaches, we review existing approaches to testing aspect-oriented systems.

As we discussed in Section III, most approaches to computing a class integration test order for an object-oriented system are graph-based. In particular, these approaches rely on an ORD to represent classes (as nodes) and inter-class dependencies (as edges). Further, some approaches (e.g., those by Kraft *et al.* [2] and Abdurazik and Offutt [15]) use edge weights to represent the cost of creating the stub that would be needed were the edge removed. In general, approaches to computing class integration test orders can be characterized by the kinds of dependencies that they model (as edges) in the graph and by the strategies that they use to remove cyclic dependencies from the graph.

Kung *et al.* [24] present the initial solution to computing a class integration test order. They construct an ORD that models the following dependencies: *aggregation*, *association*, *inheritance*, *template instantiation*, *instantiated template use*, *nested*. They further divide *aggregation* into *automatic*, *static*, and *dynamic*. Their template dependencies are specific to C++, and nested corresponds to the *ownedElement* relationship from UML class diagrams. Kung *et al.* remove the *association* edges that will "result in the minimum number of stubs." Tai and Daniels [25] present another early solution. They include three edge types in their ORD. Their solution is most notable in that it relies on node removal, rather than edge removal, to remove cycles.

Labiche *et al.* [26] noted that not all dependencies are static and thus introduced the *dynamic* edge to their ORD. This edge type accounts for dependencies introduced by dynamically dispatched (virtual) method calls. Others [27], [2] have since termed this edge type *polymorphic*, as do we in this paper. More recent work by Abdurazik and Offutt [15] considers a total of nine edge types, many of which represent dynamic dependencies. In particular, Abdurazik and Offutt use coupling measures to identify the following 10 dependencies: *association coupling*, *aggregation coupling*, *composition coupling*, *usage dependency*, *call coupling*, *global coupling*, *inheritance coupling*, *interface realization coupling*, *external coupling*, and *exception coupling*. They assign weights of five to edges representing *inheritance coupling* and *composition coupling*, and weights of one to edges representing any other dependency. Their weighting model reflects their belief that inheritance and composition are the dependencies which would require the most costly stubs should the corresponding edges be removed from the ORD.

Malloy *et al.* [27] and Kraft *et al.* [2] (which subsumes Malloy *et al.*) use a fully parameterized cost model for weighting edges in an ORD. They consider six edge types: *association*, *dependency*, *inheritance*, *composition*, *ownedElement*, and *polymorphic*. Further, they experiment with different combinations of edge weights and compare their edge removal algorithm to that of Briand *et al.* [14] (whose work subsumes earlier work by Le Traon *et al.* [13] and Briand *et al.* [28]). Kraft et al. conclude that removal of *inheritance* edges can cause the total number of necessary stubs to drop dramatically. Thus, it may be cost-effective to create a complex stub for an *inheritance* edge if it obviates the need for creating many additional stubs (though those stubs may be somewhat simple to create).

Similar to Le Traon *et al.* [13], Hewett and Kijsanayothin [29] use a structure-oriented approach to identifying edges for removal from cycles of dependencies. That is, rather than consider the type of the edge to be removed, they focus on the structure of the (cyclic) subgraph in which the edge is found. Though Hewett and Kijsanayothin are able to minimize the number of stubs using their approach, they do not consider the cost of creating each stub.

Genetic algorithms have been used for the class integration test order problem. Da Veiga Cabral *et al.* [30] have proposed a multi-objective optimization approach, where the algorithm tries to generate solution with a balance compromise between different metrics (the objectives). The metrics used are attribute complexity and method complexity that measures the number of attributes (respectively, methods) that would need to be handled in the stubs for a given solution. Assuno *et al.* [31] have investigated the usage of two multi-objective evolutionary algorithm, using four metrics: attribute coupling, method coupling, number of distinct return types, and number of distinct parameter types. Experiments on different systems showed that both algorithm are effective for the class integration test order problem.

In previous work [32], we have proposed an approach to evaluate the impact of aspects on the test cases. This approach assumes that a class has been tested before an aspect is woven and aim to statically find the set of all the test cases that are impacted by this aspect. The static analysis is close to the one we use in this paper to find the methods impacted by an aspect. In our previous work the analysis starts from a test case and try to find a reachable advice, whereas in this paper the analysis starts from an advice and compute the set of the methods that can reach this advice.

As discussed in Section II previous work on integration testing of aspect-oriented programs have focused on an incremental approach. Zhou *et al.* [4] present an approach for testing aspect-oriented programs encompassing unit testing, integration testing, and system testing. The authors argues that for integration testing an order is not important in the case of aspect-oriented programs. In this paper we advocate the use of a fine grain integration testing order that reduce the testing effort. Ciccato *et al.* [5] have discussed the difficulty of testing aspect-oriented programs, and suggested using an incremental approach to leverage the separation of concerns. Xu *et al.* [6] have proposed an approach for integration testing of aspect-oriented programs that rely on state-models. State models are finite state machines that model the behavior of the system and that are used to generate test cases. They propose an incremental approach where the state model first only models the base program. Then the state model is modified to take into account the aspects. State models could be used to generate test cases in a combined approach that uses an order where aspects and classes are interspersed.

Ré *et al.* [33] have adapted the algorithm of Briand *et al.*[14] by extending the ORD to take aspects into account. The produced ORD does only describe direct impact of the aspects and, as discussed in Section III, the algorithm uses only local information to make a decision in a strongly connected component. Colanzi *et al.* [34] have proposed a multi-objective approach for the class integration test order problem in the context of aspect-oriented programming. The two studied algorithm, as well as the four metrics, are the same used by Assuno *et al.* [31], whereas the ORD used is based on the work of Ré *et al.* [33]. The algorithms were studied on different projects and showed to be effective. The approach we present in this paper is close but uses a fitness function that takes the indirect impact of aspects into account, which, to our best knowledge, no other work does.

Several works have proposed test criteria for the integration testing of aspect-oriented programs. Test criteria are constraints that test suites should satisfy to ensure a minimal quality. Massicotte *et al.* [35] have proposed test criteria based on collaboration diagrams, control flow graph, and message trees. Lemos *et al.* [36] have proposed structural coverage criteria. This approach uses models that encapsulate both the control flow and the data flow. These models are used to define several coverage criteria for the interactions between the advices and the methods. As the goal is to check that the test cases cover all the possible interactions, these criteria can be used in an incremental approach or in a combined approach such as the one we propose in this paper. Harman *et al.* have used a search based optimization technique to generate test data for aspect-oriented programs. This technique tries to achieve structural coverage.

VII. Conclusion

In this paper we argue that the *incremental* approach of integration testing of aspect-oriented systems, where the system without aspect is tested first and then the system with the aspects is tested, induces an extra testing effort. We also argue that this testing effort can be avoided by a *combined* approach that intersperses classes and aspects.

We propose a genetic algorithm that uses information on the relations between classes and aspects, as well as information on the methods impacted by the aspects, to produce an integration testing order such that aspects are tested after the classes that they directly impact and before the classes that they indirectly impact. Such an integration order can avoid the modification of test suites for the indirectly impacted classes.

This genetic algorithm has been implemented in a tool for AspectJ programs. This implementation allowed the validation of the approach, as well as the empirical settings of the parameters of the algorithm. Experiments on the motivating example have shown that the proposed approach can produce fine grained integration test orders.

In future work we would like to apply the approach on several large size systems. Such experiments would further validate the approach, but could also be more conclusive as of the value of the weaving relationship. The experiment on the bank example has shown that a value of 10 was a minimum, but could not discriminate between the values greater than 10.

It would also be interesting to extend the fitness function with different metrics that could refine the produced integration orders. Currently, different integration orders can have the same fitness value, but it is likely that they are not equivalent, even if they are very close. A finer fitness function could discriminate between these orders and improve the algorithm.

References

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. New York, NY, USA: McGraw-Hill, 2005.

[2] N. A. Kraft, E. L. Lloyd, B. A. Malloy, and P. J. Clarke, "The implementation of an extensible system for comparison and visualization of class ordering methodologies," *Journal of Systems and Software*, vol. 79, no. 8, pp. 1092–1109, 2006.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997, pp. 220–242.

[4] Y. Zhou, D. Richardson, and H. Ziv, "Towards a practical approach to test aspect-oriented software," in *TECOS '04: Workshop on Testing Component-based Systems*, 2004.

[5] M. Ceccato, P. Tonella, and F. Ricca, "Is AOP code easier or harder to test than OOP code?" in *WTAOP'05: Proceedings of the first Workshop on Testing Aspect-Oriented Program*, 2005.

[6] D. Xu and W. Xu, "State-based incremental testing of aspect-oriented programs," in *Proceedings of the 5th International conference on Aspect-Oriented Software Development*, 2006, pp. 180–189.

[7] R. T. Alexander, "The real costs of aspect-oriented programming," *Software, IEEE*, vol. 20, no. 6, pp. 92 – 93, nov.-dec. 2003.

[8] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the systematic testing of aspect-oriented programs," Colorado State University, Tech. Rep., 2004.

[9] R. Delamare, B. Baudry, S. Ghosh, S. Gupta, and Y. Le Traon, "An approach for testing pointcut descriptors in aspectj," *Journal of Software Testing, Verification and Reliability*, vol. 21, no. 3, pp. 215–239, 2011.

[10] T. Reenskaug, "Thing-model-view-editor," Xerox PARC, Tech. Rep., 1979.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley, 1995.

[12] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Proceedings of the Workshop on Advanced Separation of Concerns*. Addison-Wesley, 2000, pp. 21–35.

[13] Y. Le Traon, T. Jeron, J.-M. Jézéquel, and P. Morel, "Efficient object-oriented integration and regression testing," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 12–25, 2000.

[14] L. C. Briand, Y. Labiche, and Y. Wang, "An investigation of graph-based class integration test order strategies," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 594–607, 2003.

[15] A. Abdurazik and J. Offut, "Using coupling-based weights for the class integration and test order problem," *The Computer Journal*, vol. 52, no. 5, pp. 557–570, 2009.

[16] L. C. Briand, J. Feng, and Y. Labiche, "Using genetic algorithms and coupling measures to devise optimal integration test orders," in *Proceedings of the 14^{th} international conference on Software engineering and knowledge engineering*. New York, NY, USA: ACM, 2002, pp. 43–50.

[17] R. Storn and K. Price, "Differential evolution a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, pp. 341–359, 1997.

[18] R. Chelouah and P. Siarry, "A continuous genetic algorithm designed for the global optimization of multimodal functions," *Journal of Heuristics*, vol. 6, pp. 191–213, 2000.

[19] Y.-W. Leung and Y. Wang, "An orthogonal genetic algorithm with quantization for global numerical optimization," *Evolutionary Computation, IEEE Transactions on*, vol. 5, no. 1, pp. 41 –53, feb 2001.

[20] T. T. Chow, G. Q. Zhang, Z. Lin, and C. L. Song, "Global optimization of absorption chiller system by genetic algorithm and neural network," *Energy and Buildings*, vol. 34, no. 1, pp. 103 – 109, 2002.

[21] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991, pp. 69–93.

[22] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Trans. Syst. Man Cybern.*, vol. 16, pp. 122–128, January 1986.

[23] K. A. De Jong, "Analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, The University of Michigan, 1975.

[24] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "A test strategy for object-oriented programs," *Computer Software and Applications Conference, Annual International*, vol. 0, p. 239, 1995.

[25] K.-C. Tai and F. J. Daniels, "Test order for inter-class integration testing of object-oriented software," *Computer Software and Applications Conference, Annual International*, vol. 0, p. 602, 1997.

[26] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand, "Testing levels for object-oriented software," *Software Engineering, International Conference on*, vol. 0, p. 136, 2000.

[27] B. A. Malloy, P. J. Clarke, and E. L. Lloyd, "A parameterized cost model to order classes for class-based testing of c ++ applications," *Software Reliability Engineering, International Symposium on*, vol. 0, p. 353, 2003.

[28] L. C. Briand, Y. Labiche, and Y. Wang, "Revisiting strategies for ordering class integration testing in the presence of dependency cycles," in *Proceedings of the 12th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 287–.

[29] R. Hewett and P. Kijsanayothin, "Automated test order generation for software component integration testing," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 211–220.

[30] R. da Veiga Cabral, A. Pozo, and S. Regina Vergilio, "A pareto ant colony algorithm applied to the class integration and test order problem," in *ICTSS'10: Proceedings of the 22^{nd} International Conference on Testing Software and Systems*, 2010.

[31] W. K. G. Assuno, T. E. Colanzi, A. T. R. Pozo, and S. R. Vergilio, "Establishing integration test orders of classes with several coupling measures," in *GECCO'11: Proceedings of the 13^{th} conference on Genetic and Evolutionary Computation*. ACM, 2011, pp. 1867–1874.

[32] R. Delamare, F. Munoz, B. Baudry, and Y. Le Traon, "Vidock: a tool for impact analysis of aspect weaving on test cases," in *Proceedings of the 22^{nd} IFIP International Conference on Testing Software and Systems (ICTSS'10)*, 2010.

[33] R. R, O. A. L. Lemos, and P. C. Masiero, "Minimizing stub creation during integration test of aspect-oriented programs," in *Proceedings of the 3rd workshop on Testing aspect-oriented programs*. New York, NY, USA: ACM, 2007, pp. 1–6.

[34] T. E. Colanzi, W. K. G. Assuno, S. R. Vergilio, and A. Pozo, "Integration test of classes and aspects with a multi-evolutionary and coupling-based approach," in *SSBSE'11: Proceedings of the third international Symposium on Search Based Software Engineering*, 2011.

[35] P. Massicotte, M. Badri, and L. Badri, "Generating aspects-classes integration testing sequences: A collaboration diagram based strategy," in *SERA '05: Proceedings of the Third ACIS International Conference on Software Engineering Research, Management and Applications*, 2005, pp. 30–39.

[36] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero, "Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for java," *Sci. Comput. Program.*, vol. 74, pp. 861–878, August 2009.