Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies

Vu Le Hanh [1], Kamel Akif [2], Yves Le Traon [1], Jean-Marc Jézéquel [1]

[1]: IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France.
{Hanh.Vu_Le, Yves.Le_Traon, Jean-Marc.Jezequel}@ irisa.fr
[2]: LAN/DTL/FT R&D Lannion, 2 av. Pierre Marzin, 22307 Lannion Cedex, France.

Abstract: The normalization of semi-formal modeling methods, such as the UML, leads to re-visit the problem of early OO integration test planning. Integration is often conducted under some incremental steps. Integration test planning aims at ordering the components to be integrated and tested in relationships with the already tested part of the system. This paper presents a modeling of the test integration problem from a UML design, then details existing integration strategies and proposes two integration strategies: a deterministic one called Triskell and an original semi-random one, based on genetic algorithms called Genetic. Strategies are compared in detail (algorithmic cost and optimization choices) and a large part of the paper is dedicated to an experimental comparison of each strategy on 6 real-world case studies of various complexities (from a "small" telecommunication software to the Swing Java library). Results show that a good modeling of this optimization problem associated with well-chosen algorithms induce a significant gain in terms of testing effort and duration.

Key word: Software Testing, Object-Oriented Modeling, UML, Test Economics, Test Cost, Integration Testing, Graph Algorithms, Stub Minimization.

1 Introduction

Design-for-testability aims at integrating design and testing in the same process, and includes the problem of test planning from early design stages. In the case of object-orientation, due to inheritance and dynamic binding, the control is no more centralized in the main encapsulation unit, namely the class. However, the testing task remains unchanged but must deal with the whole architecture to determine subtle errors. The particular problem tackled to testers by OO architectures (class and package diagrams) is the strong connectivity between their components. While integration testing is performed to find errors in component interfaces when they communicate with each others, the complexity of these architectures makes the classical integration strategies useless.
To efficiently pinpoint the errors, it is preferable to avoid the "big-bang" integration methods [1, 2]. Instead, an incremental strategy is appropriate: the more classical integration strategies are "bottom-up" and "top-down" ones. They are based on a graph representation of the system under integration that is assumed to be acyclic.

Bottom-up methods begin by testing leafs of the graph and then tests the upper levels step by step, while top-down methods begin by the top level and descends in the graph hierarchy step by step [2]. In the first case, only test drivers have to be implemented, while the top-down also needs "stubs" for simulating the lower non-integrated components. These strategies are meaningless for most OO systems, where cycles of dependencies between components often exist, as it will be shown in the case studies. The integration problem must be re-thought for OO systems. A model must be defined and strategies proposed to take into account the highly connected structure of most OO designs.

This paper focuses firstly on the problem of bridging this gap between the actual design architecture, expressed by UML class and package diagrams, and an abstract modeling of the integration test problem (in terms of test dependence graph). Then, we concentrate on the algorithmic problem of producing the best integration test plan for delay-driven projects. The main difficulties for a cost/duration-efficient integration is the minimization of the number of "stubs" to be written (cost) and the number of steps needed to achieve the integration (duration). A stub is a dummy simulator used to simulate the behavior of a real component not yet integrated. In the paper, the performance of a strategy is thus related to the number of generated stubs and the number of integration steps.

Among the many existing proposals [4, 6, 9, 10, 11, 13] that deal with this problem, we concentrate on those of David C. Kung and al. [10] and Kuo Chung Tai & Fonda J. Daniels [4] since they both detail effective and feasible algorithms. A large part of the paper is thus devoted to an experimental comparison between the performances of these approaches with respect to two original strategies (one deterministic and one genetic based algorithm). This comparison is based on the complexity of the algorithm, the number of stubs needed as well as the number of integration steps. Six real-world case studies of various scales serve as the comparison benchmark. Since the theoretical problem is NP-complete, no optimal feasible solution is existing. However, for the chosen comparison criteria, the results reveal that the various approaches are not equivalent and that the proposed algorithms are very promising.

This paper is organized as follows. In Section 2, the two-dimensional (effort/duration) problem of integration testing is detailed. Section 3 opens with some definitions about structural test dependencies and outlines the mapping from UML to a test dependence graph (with a more complete set of modeling rules than the one presented in [12]). Then, four algorithms to compute an integration test plan by minimization of the number of stubs are outlined (Section 4): the Triskell and Genetic ones correspond to the main contributions of this paper with the experimental results presented in Section 5. One of the difficulties for such comparison is to exhibit significant criteria of cost/duration performances: our assumptions and their limitations are detailed. The experimental comparison between the performances of each strategy is presented at the end of the fifth section.

2 Stub Minimization and Testing Resource Allocation

Integration testing is defined here as the way in which testing is conducted to integrate components into the system. A "component" is a stable part of the software

with a well-defined interface described wit e.g. a UML model: for sake of clarity, a component will be restricted either to a class or to its specific method in the detailed design. One of the main difficulties for a cost-efficient integration is to minimize the number of stubs to be written. Integration testing duration also depends on the number of testing resources available for performing the integration.
The problem of integration planning is a two dimensional one. One dimension of the problem is the allocation of the available testing resources (or testers) to the integration tasks, the second dimension concerns the minimization of the effort to create the stubs. For sake of clarity, we consider in this paper that the number of stubs to be created is a good indicator of the testing effort: we just mention that the nature of the stub and its complexity could be taken into account by *ad-hoc* adaptations of the presented algorithms. To deal with this two-dimensional problem, each strategy focus on one of these aspects first (both dimensions are NP-complete). Depending on the analysis of the problem, the integration strategies may either be allocation-first or stub-minimization-first guided.

2.1 Stub Minimization

Let us to define what is a stub: if a component C_1 uses one or more service(s) of another component C_2, we say C_1 ***"depends"*** on C_2. During the incremental integration process, when C_1 is integrated, if C_2 has not already been integrated, we have to simulate the services of C2. Such a simulator is usually called a ***"stub"***.
In this paper, we model two types of stubs: specific stubs and realistic ones (see Figure 1).

– *A specific stub* simulates the services for the use of a given client only. This kind of stub forces the caller component to used a predefined calling sequences since the stub provides "canned" outcomes for its processing. In that case, the stub is specific to a particular caller component, and as many specific stubs have to be created as they are caller components.
– *A realistic stub* simulates all services that the original class can provide. In that case, the stub works whatever the caller component is using it.

Note that realistic stubs can be obsolete (but reliable) implementations of stubbed components (see Figure 1) as well as an available library that would be replaced by a dedicated component later.
A stub is not a real component and will not be used in the final product. Thus, we have to minimize the effort to create stubs. In particular, if we assume that every stub requires the same effort to be created, then the minimization of stub creation effort is synonymous with the minimization of the stub number. This assumption could be easily relaxed by associating complexity values to stubs, corresponding to their creation effort. No stub is needed when the dependencies between components in the system generate no cycles: integration strategies will thus differ mainly by the way they detect cycles and the criterion used to break them.

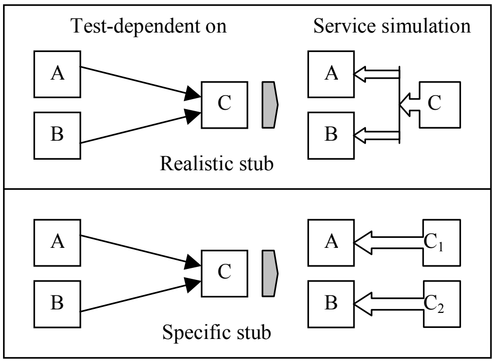


Figure 1. Realistic/specific stubs

2.2 Testing Resource Allocation

Let us define a ***"tester"*** as a testing team of a given fixed size. It represents a given unit of testing effort that can be allocated to a testing task. Let us make the following assumption to help presenting the test repartition strategy: a "tester" needs one time unit or "***step***" to integrate a component to the system and test it. This notion of step can easily be adapted by giving a weight to each component corresponding to its estimated testability. In terms of time measurement, the integration is a discrete process and not a continuous one: that is the reason why step notion is close to the real integration activity. However, the mapping of a "step" into a real-time measurement is not studied here since we only try to capture the following information: being given a system, which strategy requires the less time for achieving integration?

By another way, the question of testing resources allocation is: being given n testers, what is the best way to allocate integration testing tasks to minimize the number of steps of the integration?

Both problems, testing resource allocation and stub minimization, always go together. When we allocate a component to a "tester", the tester may have to create a stub, especially in the object-oriented paradigm where components can depend, directly or indirectly, on each other and vice-versa. Depending on the analysis of the problem, the integration strategies may either be allocation-first guided or stub-minimization-first guided. In the literature, David C. Kung and al. [10], Kuo Chung Tai and Fonda J. Daniels [4] proposed strategies to allocate the testing resources first but they did not explicitly deal with the dimension of stub minimization.

The third (Triskell [12]) and fourth (Genetic) ones correspond to our propositions. We argue that the best way to deal with this two-dimensional problem (both dimensions are NP-complete) is to first guide integration by minimizing stubs and then allocating resources.

3 From UML to Test Dependence Graph (TDG)

In this section, we recall the notion of test dependencies and its associated model called Test Dependence Graph (TDG). This model has been introduced in [7, 12] but the part of model concerning regression testing is simplified here. Indeed, implementation parts from specification/contractual ones are not distinguished in the context of integration testing. The presented model has also been extended to capture the polymorphic dependencies (through transitive relationships) as well as the nature of the dependence (Aggregation, Association, Inheritance or Implementation). We then obtain a general model on which all the published integration strategies can be implemented: Kung's ORD model (that is also the underlying model used by Tai & Daniels) is a particular case of the TDG. This common basis for modeling makes comparison possible.

Definitions

– *Test Dependence Graph (TDG):* It is a directed graph whose vertices represent components (classes and/or included methods, depending on the detail level of the design) and directed edges represent test dependencies. In a TDG, loops may occur because components may be directly or indirectly test dependent from each other.
– *Test dependence levels:* Depending on the level of detail in the design, we may define test dependencies between classes or between methods of classes (in case of high detail level). We distinguish three levels of test dependencies:
    1. *Class-to-class*: It is the dependence level that can be induced from a design model, as soon as a stable class diagram is available. A vertex in a TDG models a class. An edge of TDG links these vertices.
    2. *Method-to-class*: If a method m has an object of a class C declared in its signature, a method-to-class dependence exists between this method and this class. In TDG, we model this method and class by vertices. These vertices are linked with an edge.
    3. *Method-to-method:* This dependence can be inferred only by analyzing the implementation body of a method. If method m_1 calls method m_2, a method-to-method dependence exists between these methods. In the TDG, we model these methods by two vertices and directed edges connecting m_1 to m_2 and all redefinitions of m_2 in subclasses of class containing m_2 (dynamic binding).

From a UML class diagram, only class-to-class and method-to-class test dependencies can be inferred to build a TDG. Using information available in the UML dynamic diagrams, some method-to-method test dependencies can be inferred. If source code is available, then all test dependencies can easily be extracted to build a TDG.

Concerning only the most general class-to-class test dependencies, the TDG can be extracted from a design model such as a UML description of an OO system. The way the TDG is build is a (safe) over-estimation of the actual dependencies. In the TDG, an arrow from B to A means that "B is test dependent on A". If it is possible, A should be tested before B. When more detailed information is available on a system (through a detailed model or extracted from the code) we can produce a more accurate

TDG at the granularity of methods as explained in [12]. Since the actual strategies – except for Triskell one – do not take into account this level of detail, this part of the model is not used for comparison

Three main rules are used to map the UML model into a TDG as presented in Figure 2. Being given two components: A and B:

– If B inherits (derives from) A, A is test-dependent on B by an inheritance dependence. The edge that connects vertex A to vertex B is labeled “I” in the TDG.
– If A is a composite (an aggregate) of B, A is test-dependent on B by an aggregation dependence. The A to B edge is labeled “Ag” in the TDG.
– If A is associated or depends on B, A is test-dependent on B by an association dependence. The A to B edge is labeled “As” in the TDG.

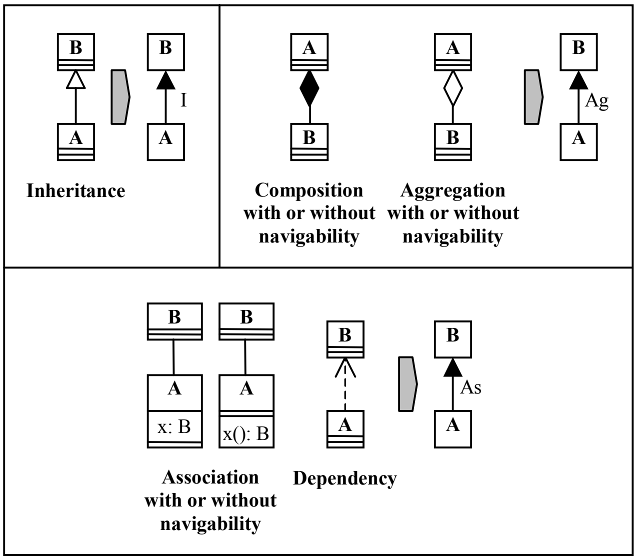

B
A
B
I
A
Inheritance
A
B
A
B
B
Ag
A
Composition with or without navigability
Aggregation with or without navigability
B
A
x: B
B
A
x(): B
B
A
B
As
A
Association with or without navigability
Dependency

Figure 2. UML to TDG: Class-to-class edges: Main transformations

To deal with polymorphism dependencies (the rules (a) and (b) in the left part of Figure 3), we choose to add extra edges to the TDG. If a component A is test-dependent on a component B by an aggregation dependence or by an association dependence, A is test-dependent by an aggregation dependence or by an association dependence (respectively) on all components derived from B.

In the right part of Figure 3, we take into account abstract components and interface ones that cannot be instantiated. In the figure, rules (c) and (d) must be applied first, and then the third rule (e) for deleting the vertex corresponding to the abstract class B.

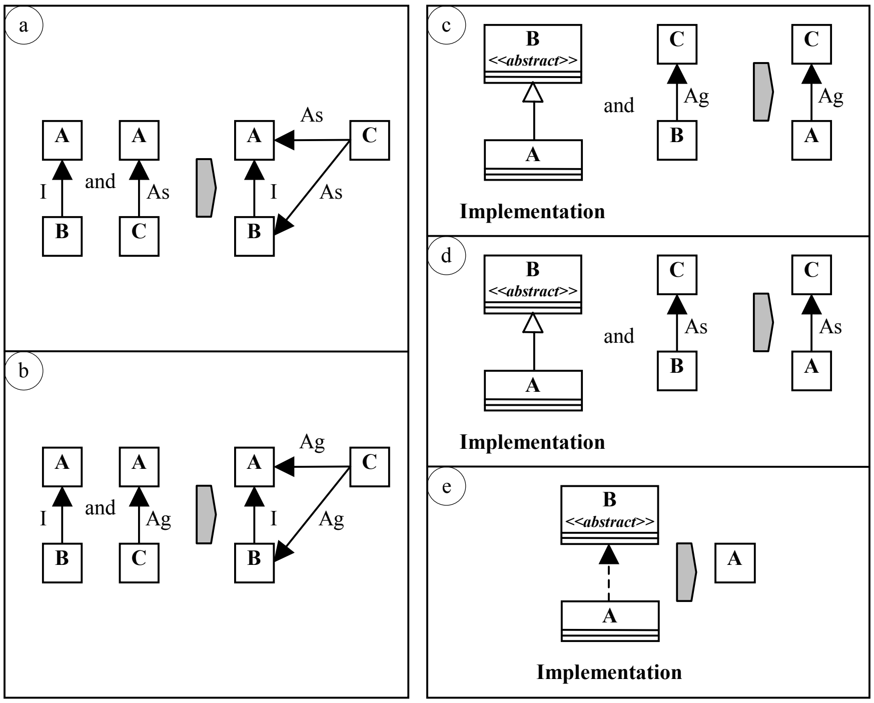


Figure 3. Polymorphic and abstract classes transformations rules

4 Integration Strategies

In this section, we present three deterministic strategies (Kung's, Tai-Daniels' and Triskell) and one semi-random strategy based on genetic algorithms (Genetic by Akif Kamel). All solutions presented here are based on a TDG. If a set of components is included into a cycle of dependencies, we say these components belong to the same ***Strongly Connected Component*** or ***SCC.*** The main difference between the various strategies lies in the way that cycles are detected and then broken.
To illustrate these strategies, we use a small example of TDG (see Figure 4). The outcome of each strategy will be presented step by step. Even if some algorithms may appear complex and not intuitive, we choose to detail the application of each strategy precisely on this small example to make the experiments repeatable and the paper self-contained.
Several points have to be noticed:

– Each strategy is presented in three parts, (i) a stub choice step, (ii) a resource allocation step and (iii) its illustration with the TDG of Figure 4.
– Since a stub is a dummy component, when a component A needs a stub for a component B during its integration, the integration will not be achieved until the component A has been tested with the real component B. In the result of our example, we underline this step of testing with the real component B by putting it in (parenthesis).

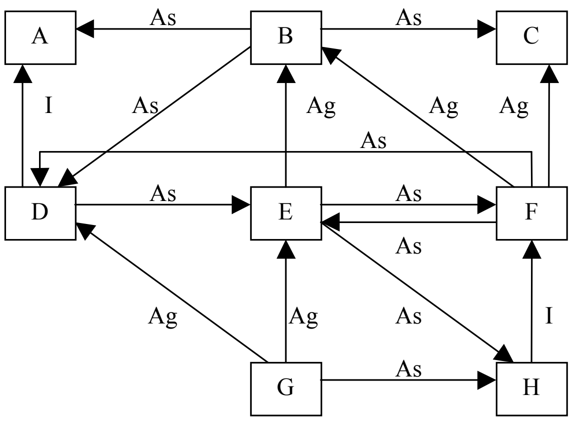


Figure 4. An example of TDG

- The result of an integration test strategy is a partial order tree, each vertex representing a component. For sake of conciseness, this partial order tree has been flattened as follows: when a choice between equivalent vertices is done, we use the ASCII code order of the vertex labels. For example, between vertices C and E, we will firstly choose C (topological sort).
- Stubs are needed because of the existing SCCs. Each strategy applies a proper criterion to remove some edges and break the SCC. Any successor from a removed edge is stubbed. Dotted arrows represent such edges and gray background boxes represent stubs

4.1 Kung and al.'s Strategy [10]

To choose a stub, Kung argues that an association relationship always exists inside a cycle and this kind of relationship is the weakest of three kinds: Inheritance, Aggregation and Association. To break a cycle, one of its association relationships is removed. The testing resource allocation procedure is based on the "***height***" of vertices in the TDG, i.e. the maximum number of vertices included in paths from the considered vertex to a leaf vertex. Then a leaf vertex has a height value of 1.
Although Kung gives a way to choose a stub, the problem of stub minimization is not explicitly taken into account since the strategy is first allocation-guided. This algorithm is illustrated in Figure 5.

(i) Stub choice:

- Search all SCCs using transitive closure.
- Assign a height for each SCC. This height is called ***"major level"***.
- For each SCC, recursively remove association edges until there are no more cycles.
- Successors of removed association edges are stubbed. They are specific stubs.

(ii) Resource allocation:

- Without taking into account the edges removed in step 1, a height value called ***"minor level"*** is assigned to each vertex of each SCC. This value allows defining a partial order into a SCC.
- The vertices allocation is based on the "***pair (major level, minor level)***" that defines a partial order:

  smaller: $(x_1,y_1) < (x_2, y_2) \Leftrightarrow (x_1 < x_2) \vee (x_1 = x_2 \wedge y_1 < y_2)$
- The vertex with smaller pair (major level, minor level) will be allocated earlier. If two vertices have a same pair (major level, minor level), choose any one to stub.

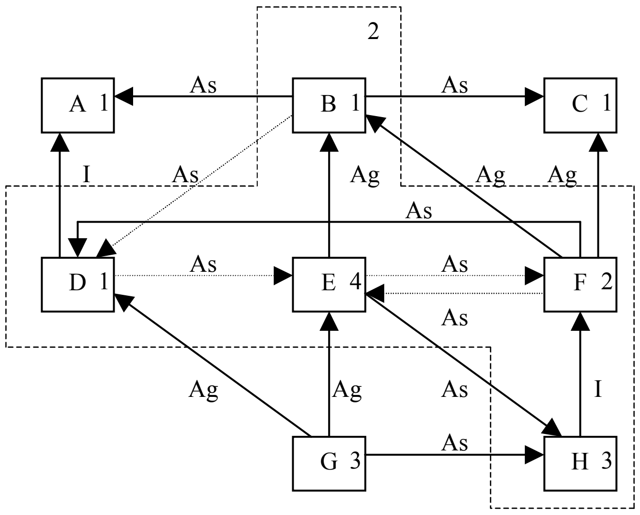


Figure 5. Kung and al.'s strategy

(iii) Illustration:

Table 1 and 2 present the application of Kung and al.'s strategy on the TDG of Figure 4. Table 1 underlines the SCC order and corresponding stub choice: the strategy requires 4 specific stubs (dotted arrows) and 3 realistic stubs (gray background boxes).

Table 1. The SCC and the edges to remove

SCC	Major Level	Edge to remove
{A}	1	∅
{B D E F H}	2	{BD, DE, EF, FE}
{C}	1	∅
{G}	3	∅

Table 2 shows the pairs (major level, minor level) of all vertices in Figure 4 and their integration order. The order in parenthesis concerns the re-allocation step: stub replacement by the real component for final integration and testing. The number of integration steps is 11 (the maximum order).

Table 2. Major level, minor level and integration order

Vertex	Major level	Minor Level	Integration order	Vertex	Major Level	Minor level	Integration order
A	1	1	1	E	2	4	8
B	2	1	3, (5)	F	2	2	6, (10)
C	1	1	2	G	3	1	11
D	2	1	4, (9)	H	2	3	7

4.2 Tai-Daniels' Strategy [4]

To choose a stub, Tai and Daniels adapt their strategy from Kung's argument concerning association priority in the allocation step. To break a given cycle, some association relationships are removed. For the resource allocation procedure height of vertices in the TDG is used to define an order. This algorithm is illustrated in Figure 6.

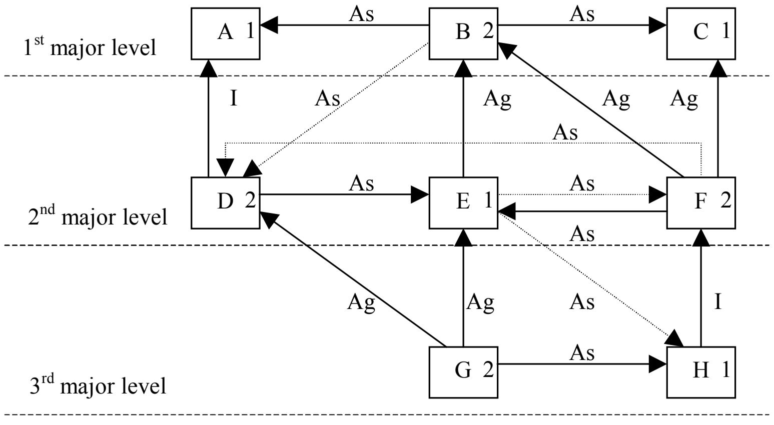


Figure 6. Tai-Daniels' strategy

(i) Stub choice:

- Assign a height for each vertex, using a depth-first algorithm, without taking into account the association relationships. This height is called ***"major level"***.
- All successors of association edges from smaller major level to greater major level are stubbed.
- Successors of removed association edges are stubbed. They are specific stubs.
- In each major level, association edges are removed if they are in a cycle and their successor is a stub.
- If there is still a cycle, a weight is assigned to each edge of this cycle. The weight of edge "e" is calculated:
  weight (e) = (number of existing edges, incoming to predecessor of e, from other vertices in the same major level with predecessor of e) +

(number of existing edges, outgoing from successor of e, to other vertices in the same major level with successor of e).

- Remove successively edges with respect to their priority weights until there are no more cycles.

(ii) Resource allocation:

- Do not take into account the edges removed in the first part, assign a local height for each component in each major level using a depth-first algorithm. This number is called ***"minor level"***.
- Allocate vertices according to the order given by pairs (major level, minor level). The vertex with smaller pair (major level, minor level) will be firstly allocated. If two vertices have same priority, choose any one to allocate first.

(iii) Illustration:

Table 3 and Table 4 present the result of Tai-Daniels's strategy applied to Figure 4.

Table 3. Major level and inter-level edges removed

Major Level	Vertices	Edge to remove
1	A, B, C	BD
2	D, E, F	EH
3	G, H	

Tai-Daniels' strategy requires 2 specific stubs (dotted arrows) and 2 realistic stubs (gray background boxes) for inter-level dependencies. The number of integration steps is 11 (the maximum order).

Remarks:

The edge FD in 2nd major level participates in the cycle (D E F) and D is a stub for upper major level (1st major level). So, to break this cycle, we remove this edge (3rd specific stubs). In 2nd major level, there is still a cycle (E F) and no E, neither F is ready a stub. Hence, to break this cycle, we have to use the edge weight.

The weight (EF) = 2 + 1 = 3, weight (FE) = 1 + 1 = 2. Between two edges, EF and FE, weight (EF) > weight (FE) and we choose to remove the edge EF (4th specific stubs). The successor of this edge is stubbed. F is the 3rd realistic stub.

Table 4. Major level, minor level and integration order

Vertex	Major level	Minor level	Integration order	Vertex	Major level	Minor level	Integration order
A	1	1	1	E	2	1	4, (8), (10)
B	1	2	3, (6)	F	2	2	7
C	1	1	2	G	3	2	11
D	2	2	5	H	3	1	9

4.3 Triskell Strategy

Triskell strategy is a two-part strategy: the first part corresponds to the stub minimization problem while the second focus on testing resource allocation. The method does not take into account the type of relationship between components as a priority. To break a cycle, the vertex that participates in as many cycles as possible is stubbed. The underlying argument is that the nature of the relationship is less important than the effort to stub any relationship. So the algorithm is first stub-minimization guided. The nature of the relationship (association, inheritance, aggregation) is also taken into account only as a second priority, using Kung's argument (association first). The testing resource allocation procedure is based on the depth of vertices in TDG, i.e. the maximum number of vertices included in paths from the considered vertex to a root vertex. Then a root vertex has a depth value of 1. This algorithm is illustrated in Figure 7.

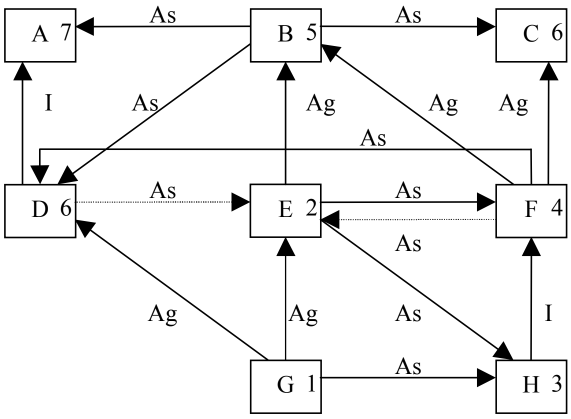


Figure 7. Triskell strategy

(i) Stub choice:

– Progressively stub the vertices, which participate in as many cycles as possible, using Tarjan's algorithm [15, 12]. Let *vertex_2_cycles (vertex)* be the function that results in the set of cycles to which the vertex belongs. When a vertex *v* is stubbed, each cycle of the *vertex_2_cycles (v)* set is broken and the incoming edge of v in this cycle is removed to create a specific stub. For a given vertex, there are as many specific stubs as the number of independent cycles traversing this vertex.
– When two vertices belong to the same number of cycles, according to Kung's argument, we stub the vertex which its incoming association edges belong to more cycles than the other. In case of equality, the next criterion consists of selecting the vertex with the lower incoming degree in the SCC. This criterion minimizes the number of specific stubs. Finally, if needed, an arbitrary order is taken, e.g. the vertex identifier ASCII code order

(ii) Resource allocation:

– Do not take into account the removed edges in the first part, assign a depth for each vertex
– Allocate vertices with priority of maximum vertex depth: bigger depth, earlier allocation. If two vertices have the same depth, we choose the vertex with smaller identification.

(iii) Illustration:

Table 5 shows all cycles to which each vertex belongs. The vertex E is traversed by the maximum number of cycles. This vertex is chosen to be stubbed (1st realistic stub). The incoming edges of E in these cycles are DE and FE. They are broken to create 2 specific stubs. Table 6 presents the depth of each vertex and its integration order. The number of integration step is 10.

Table 5. vertex_2_cycles results

Vertex	vertex_2_cycles (vertex)	|vertex_2_cycles (vertex)|
A	-	0
B	(B D E), (B D E F), (B D E H F)	3
C	-	0
D	(B D E), (B D E F), (B D E H F), (D E F)	4
E	(B D E), (B D E F), (B D E H F), (D E F), (D E H F), (E F), (E H F)	7
F	(B D E F), (B D E H F), (D E F), (D E H F), (E F), (E H F)	6
G	-	0
H	(B D E H F), (D E H F), (E H F)	3

Table 6. Depth and Test Order

Vertex	Depth	Test Order	Vertex	Depth	Test Order
A	7	1	E	2	7
B	5	4	F	4	5, (8)
C	6	2	G	1	10
D	6	3, (7)	H	3	6

4.4 Using Genetic Algorithms for Stub Minimization

Since the question of integration planning is a two-dimensional NP-complete problem, semi-random optimization algorithms have been studied. We present here an original application of genetic algorithms to this problem. On one hand, genetic algorithms (GAs) [5, 14] are applied to reduce the search space and direct it to reach a good solution. On the other hand, they allow escaping routes out of local optimization. They are widely used in many areas for problem optimization and are

efficient for solving NP-complete problems. In our study we use them to search the first dimension of integration problem, the minimization of stub number.
GAs are iterative non-deterministic algorithms, one important characteristic of these algorithms is their generality and the ease of implementation. They require a fitness function, a mechanism to traverse the search space (via crossover and mutation operations) and a suitable solution representation (by individual and population).
In our strategy, the fitness function is the number of stubs, each individual is an integration order and the population is a set of integration orders. Our strategy consists of:

(i) Stub Choice:
Choose a maximum number of iterations.
For each SCC:
- Randomly generate some initial individuals (i.e. integration order) for the initial population P (0).
- Compute the fitness function (stub number) for each individual in the current population P (t).
- Choose two individuals with the best result of the fitness function.
- Generate P (t+1) from two chosen individuals via genetic operators: crossover and mutation.
- Repeat from the 2nd step to the end until the maximum number of iterations is reached.

(ii) Resource allocation:
Similar to Triskell algorithm.

(iii) Illustration:
We use two genetic operators:
- The mutation operator ***MU (x, y, individual)***: it exchanges the *x-th* element and the *y-th* element of *individual.*

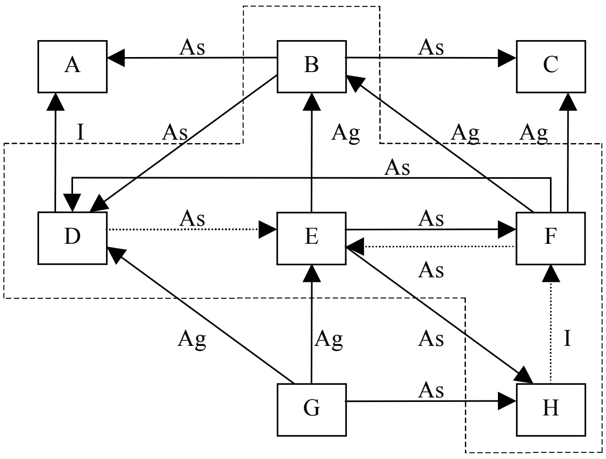


Figure 8. The SCC

– The crossover operator ***CR (x, individual1, individual2):*** we use the partially mapped crossover PMX [14]. The new generated individual I_1 contains ordering information partially determined from its parents *individual1* and *2*. A random cut point *x* is chosen in both parents, the entire right part of *individual2* is copied in the new individual I_1. If an element of the left part of *individual1* is not present in I_1 then it is added to I_1. If this element exists in I_1, then its position in I_1 is determined and the element from *individual1* corresponding to this position is copied in I_1.

Both operators use one random function to define the position cut point. The number of iterations and the order for executing the genetic operations is also randomly chosen. Another operation ***STOP (Population)*** chooses the best individual in *Population* and stops the procedure. In case of equality, the individual which i-th element has smaller ASCII code.

Table 7 shows a manipulation of the SCC {B D E F H} with the initial integration orders are I_0 = {F E H D B} and I_1 = {E B D F H}. After 3 steps, we obtain the integration order (D, B, H, F, E). We stop and the integration order for whole TDG in Figure 4 is {A, C, D, B, H, F, (H), E, (D), (F), G}. The number of step is 11. The number of realistic stubs is 2 (E and F) and of specific stubs is 3 (DE, HF and EF).

Table 7. Apply of Genetic Algorithm with Figure 4

Iteration	Population	Stub number *(fitness function)*	Genetic operation	Result
0	I_0 = {F E H D B} I_1 = {E B D F H}	4 4	MU (1, 5, I_1)	I_2 = {H B D F E}
1	I_0, I_2	4, 2	CR (3, I_0, I_2)	I_3 = {D B H F E} I_4 = {H E F D B}
2	I_2, I_3	2, 2	STOP (P_2)	I_3 = {D B H F E}

5 Case Studies

We choose to present six real-world case studies. One is from the telecommunication field and five others are in the domain of software technology ([1]).

1. A Telecommunication Switching System: Switched Multimegabits Data Service (SMDS) is a connectionless, packet-switched data transport service running on top of connected networks such as the Broadband Integrated Service Digital Network (B-ISDN), which is based on the Asynchronous Transfer Mode (ATM). A detailed description of an SMDS server design and implementation (totaling 22KLOC) can be found in [12]. The class-diagram studied here is composed of 37 classes implementing the core of the switch, with 72 connects.
2. Part of an Object-Oriented Compiler: The GNU Eiffel Compiler is a free open-source Eiffel compiler distributed under the terms of the GNU General Public

[1] All the benchmark input TDGs can be downloaded from http://www.irisa.fr/testobjets/testbenchmark

License as published by the Free Software Foundation. It is available for a wide range of platforms. The current distribution (available from http://SmallEiffel.loria.fr) includes an Eiffel to C compiler, an Eiffel to Java bytecode compiler, a documentation tool, a pretty printer and various other tools, with their sources (all in all, around 70KLOC). Its UML class diagram is available in MDL, PDF or Postcript format at http://www.irisa.fr/pampa/UMLAUT/smalleiffel.[mdl|pdf|ps]). The total size of this compiler is more than 300 classes but we dealt with its core only, totaling 104 classes with 140 connects.

3. InterViews graphic library: This case study is from Kung's article [10]. Its size is 146 classes and 419 connects.
4. Pylon library (http://www.eiffel-forum.org/archive/arnaud/pylon.htm). It is an Eiffel library for data structures and other basic features that can be used as a foundation library by more ambitious or specialized Eiffel libraries. Its size is 50 classes with 133 connects.
5. Base classes of Java 2 Platform Standard Edition Version 1.3 (http://java.sun.com/j2se/1.3/docs/api/index.html). Its size is 588 connected classes by 1935 connects.
6. Package "Swing" of Java 2 Platform Standard Edition Version 1.3 (http://java.sun.com/j2se/1.3/docs/api/index.html). Its size is 694 classes with 3819 connects.

Difficulties of the Study and Experimental Environment

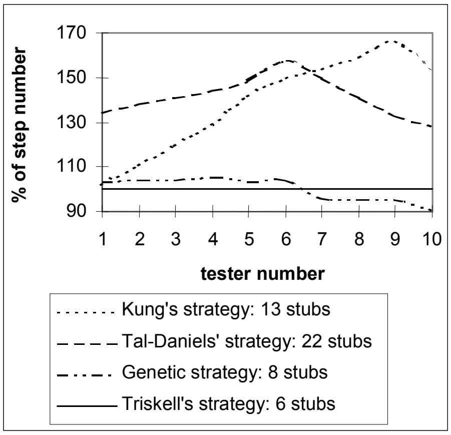
Several remarks must be done to understand the scope and limitations of the study, and provide a stable basis for replication. The validity of the experiments depends mainly on the approximation of the cost of stubs construction. Ideally, the strategies should be compared not only in terms of stub number, but also in terms of stub complexity. Kung's classical approach argues that the complexity of the stubs depends on the nature of the stubbed relationship, since "association relation represents the weakest coupling between the two related classes" while "the other two relations, namely, inheritance and aggregation, involve not only control coupling, but also code dependence and data coupling" (see [10], p. 34). However, the complexity of the integration also depends on the complexity of the functionality to be simulated, e.g. number and complexity of methods of the stubbed class. Stubbing an inherited "empty" class (e.g. containing only attributes) may be much more easier than stubbing a provider class involving many possible complex method calls. In contradiction with Kung's argument, the practice show that cycles may occur with only inheritance and aggregation relationships. In conclusion, since there is no consensual basis, we choose to consider that nothing can be decided at class level. In the case studies, Triskell strategy mainly select association edges: 1 aggregation is stubbed for Pylon case study, 4 aggregations for java library and 2 aggregations and one inheritance for Swing. For the others, only association edges are selected.
Building the TDG for SMDS, Pylon and Small Eiffel cases studies was not automated, while we have implemented a builder program to extract from the Java and Swing libraries the dependencies. This little tool uses the introspection Java mechanism.

Results

We only display the detailed results for two case studies (InterViews and Swing) in Table 8 and Table 9. The first columns of the tables list the number of testers (from 1 to 10). The other columns list the number of steps needed to integrate all the components for each strategy. A gray background marks the best value. In addition, the last line of the tables gives the number of stubs we have to create to break cycles in the TDG. This number of stubs is not dependent on the number of testers for the presented strategies.
Based on the table results, we display in Figure 9 the relative efficiency (in terms of percentage) in comparison with the result of Triskell strategy. In these figures, the "% of step number" values correspond to the ratio of the number of steps required for the strategy under consideration and the result of Triskell strategy.

Table 8. Results with InterViews graphic library case study

testers number	Kung	Tai-Daniels	Genetic Algorithm	Triskell
1	155	204	157	152
2	84	105	79	76
3	61	72	53	51
4	49	55	40	38
5	44	46	32	31
6	39	41	27	26
7	37	36	23	24
8	35	31	21	22
9	35	28	20	21
10	32	27	19	21
stub number	13	22	8	6



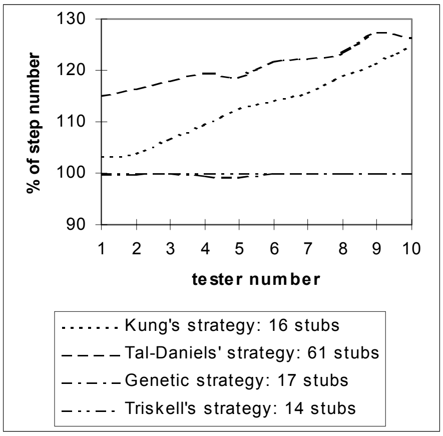


Figure 9. Relative results with InterViews (left) and Java Swing (right)

Table 9. Results with Java Swing library case study

testers number	Kung	Tai-Daniels	Genetic Algorithm	Triskell
1	740	826	715	717
2	373	418	358	359
3	255	282	239	239
4	197	215	179	180
5	162	171	143	144
6	137	146	120	120
7	119	126	103	103
8	107	111	90	90
9	97	102	80	80
10	90	91	72	72
stub number	16	61	17	14

The algorithms perform quite differently over the various case studies but main trends appear that will be discussed. Table 10 and Table 11 summarize the whole set of results obtained with five testers. The first table lists the number of stubs for each case study that was found by each strategy. The second table lists the number of steps for five testers, depending on strategy and case study. A gray background marks the best result in each line of the table.

Comparison of Algorithms and Comments

In [10], Kung et al. wrote that, according to their strategy, 8 stubs are required for the InterViews graphic library. Here, we get 13 stubs since Kung's algorithm does not specify the order of removed association relationships in the SCC. Here, we took the ASCII code order of the vertices. With another order we get 7 stubs, that is a better result than Kung with the same algorithm: due to this unspecified part, the algorithm results are not stable (in that case between 7 and 13 stubs). This remark reveals that the chosen order is very important and highly modifies the stub minimization efficiency.

Table 10. Stub number for each case study

Case studies	Kung	Tai-Daniels	Genetic	Triskell
SMDS, 37 classes, 72 connects	11	14	13	9
SmallEiffel, 104 classes, 140 connects	4	10	6	1
InterViews, 146 classes, 419 connects	13	22	8	6
Pylon, 50 classes, 133 connects	6	9	3	3
Java, 588 classes, 1935 connects	9	55	8	7
Swing, 694 classes, 3819 connects	16	61	17	14

Table 11. Step number for each case study (allocate for five testers)

Case studies	Kung	Tai-Daniels	Genetic	Triskell
SMDS, 37 classes, 72 connects	17	17	13	13
SmallEiffel, 104 classes, 140 connects	29	28	23	22
InterViews, 146 classes, 419 connects	44	46	32	31
Pylon, 50 classes, 133 connects	17	23	12	12
Java, 588 classes, 1935 connects	152	199	124	123
Swing, 694 classes, 3819 connects	162	171	143	144

Kung used a transitive closure to compute the SCCs. The complexity of this algorithm is $O(n^3)$ with n is the number of components. It requires much more time in comparison to other strategies that are based on depth-first algorithms: it can neither be used in a realistic large industry project nor for small systems at a detailed level of analysis (method level). On the contrary, Triskell is based on an adaptation of Bourdoncle's algorithm [3]. This algorithm itself is based on Tarjan's algorithm [15] (linear with the number of vertices). Then, the global algorithmic complexity is close to O(n) – the pathological unrealistic graph where there are as many SCCs as they are vertices may lead to a $O(n^2)$ complexity.
Clearly, Tai-Daniels' strategy is the less efficient one both for stub minimization and integration duration. The reason of this disappointing behavior can be illustrated with the simple example given in Figure 10: the test order is (A, B, C, D, A) with one stub (D). In Kung's strategy and in our approaches, no stub is needed and the best order (B, D, A, C) is obtained. The difference between Kung and Triskell strategies is mainly due to the fact that Kung's strategy criterion is local (association first) while Triskell algorithm globally aims at reducing the number of stubs. We already discussed the reason we cannot compare the complexity of stubs at a class level of detail: all algorithms can be adapted to take into account this inherent complexity, but no objective comparison basis would have been available in that last case.

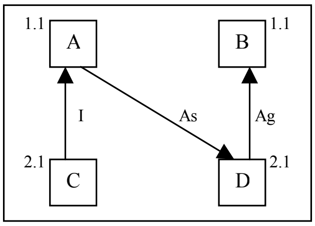


Figure 10. With Tai and Daniels' strategy, D is stubbed to test A

The behavior of the genetic algorithm is promising, but seems to be less efficient than the deterministic optimized Triskell algorithm. Detailed results in annex show that it obtains sometimes a better result than Triskell. The main advantage of the Genetic strategy is that it can more easily be adapted to take into account the complexity of components, without extra computational cost. To conclude, Kung strategy behaves

well but the computational cost is prohibitive for real projects. Tai-Daniels analysis of the problem is really pertinent but the strategy results are disappointing. Triskell is close to Kung but doesn't make any difference between stub types for determining a better result and Genetic fitness function should maybe be improved to be more efficient.
As qualitative result, we may say that integration strategies are mature enough to be really applied to large-scale projects for saving time and cost. An interesting viable, alternative to stub minimization, consist of performing big-bang integration only on the SCCs in all the cases in which their cardinality is small. The argument is that a little SCC identifies a coherent set of classes that can be tested in one time (big bang). Ideally, such a solution would let the decision of breaking a SCC to the tester. In the best case, no stub would be needed.
The assertion that cycles very rarely occur in an OO design is an opinion that is contradicted by all of ours experiments: most OO systems are highly connected and even well-written ones[1] include some of hard-to-test cycles. For instance, the Java library generates about 8.000 cycles that are broken with 7 stubs. A correlated consequence of our studies would be to pinpoint large cycles of dependencies global to several packages (since cycles into a package are acceptable).

6 Conclusion

The paper has dealt with an important issue in testing of object-oriented systems: the definition of a cost-effective integration testing process. The two-dimensional aspect of the problem of integration planning has been presented (stub minimization and testing resource allocation) as well as a model adapted to a comparison of four possible strategies. The rules for building this model from the UML have been given and the algorithms explained in detail. We have presented an empirical comparison of the performance of these algorithms to compute an integration test plan. This comparison was conducted on six real-world case studies. Both theoretical considerations (on algorithmic complexity and stub selection criteria) and experimental results allow us to differentiate these strategies. A lesson learnt from this comparison is that Triskell and Kung's strategies (the latter using Tarjan's algorithm to determine SCCs) are directly applicable on real industrial projects. The next qualitative step for improving integration testing would concern criteria to distinguish the cost of a stub. Other future work concerns on interactive strategies that combine big-bang and incremental integration. The creation of stubs could be reduced when the set of strongly connected classes corresponds to a well-defined subset of system functionality that can be reasonably tested in one block.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful and constructive comments.

[1] Java Swing is often highly pointed for its very elegant OO designs

References

1. B. Beizer, "Software testing techniques," Van Norstrand Reinhold, 1990. ISBN 0-442-20672-0.
2. Robert V. Binder, "Testing Object-Oriented Systems, Models, Parterns and Tools", Addison Wesley, First printing, October, 1999, ISBN 0-201-80938-9
3. F. Bourdoncle, "Efficient Chaotic Iteration Strategies with Widenings", Proc. of the International Conference on Formal Methods in Programming and their Applications, Lecture Notes in Computer Science 735, Springer-Verlag (1993), 128-141, ISSN 0302-9743.
4. Kuo Chung Tai and Fonda J. Daniels, "Interclass Test Order for Object-Oriented Software," Journal of Object-Oriented Programming (JOOP), v 12, n 4, July-August 1999, 18-35, ISSN: 0896-8438.
5. D. E. Goldberg, " Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989. ISBN: 0-201-15767-5
6. Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick, "Incremental Testing of Object-oriented Class Structures," Proceedings, 14th International Conference on Software Engineering, May 1992. IEEE Computer Society Press, Los Alamitos, California. 68-80. ISBN 0-7695-0915-0.
7. Thierry Jéron, Jean-Marc Jézéquel, Yves Le Traon and Pierre Morel, "Efficient Strategies for Integration and Regression Testing of OO Systems", In proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE'99), November 1999, Boca raton (Florida), 260-269, ISBN 0-7695-0807-3.
8. Jean-Marc Jézéquel, "Object Oriented Software Engineering with Eiffel," Addison-Wesley, mar 1996. ISBN 1-201-63381-7.
9. Paul C. Jorgensen and Carl Erickson, "Object-Oriented Integration Testing" Communications of the ACM, v 37, n 9, September 1994, 30-38, ISSN: 0001-0782.
10. David C. Kung, Gao, Jerry, Chen, Cris., "On Regression Testing of Object-Oriented Programs," The Journal of Systems and Software. v 32 n 1, Jan 1996, ISSN: 0164-1212.
11. Y. Labiche, P. Thévenod-Fosse, H Waeselynck and M.H. Durand, "Testing Levels for Object-Oriented Software". In proc of ICSE' 2000, June 2000, Limerick (Ireland) 138-145, ISBN 1-5811-3074-0.
12. Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel and Pierre Morel, "Efficient OO Integration and Regression Testing", IEEE Transactions on Reliability, v.49, n 1, March 2000, 12-25. ISSN 0018-9529.
13. John D. McGregor and Tim Korson, "Integrating Object-Oriented Testing and Development Processes," Communications of the ACM, v 37, n 9, September 1994, 59-77, ISSN: 0001-0782.
14. S.M. Sait, H.Youssef, "Iterative Computer Algorithms with Applications in Engineering Solving Combinatorial Optimization Problems", IEEE COMPUTER SOCIETY 1999.
15. R. Tarjan, "Depth-first search and linear graph algorithms", SIAM J. Comput., v.1, n 2, June 1972, 146-160, ISSN 1064-8275.
16. James Rumbaugh and Ivar Jacobson and Grady Booch,"The Unified Modeling Language Reference Guide",Addison-Wesley, 1998, ISBN 0-201-3099-8.