Efficient Strategies for Integration and Non-regression Testing of OO Systems

I. Introduction

Testing is becoming one of the key-aspects of OO methodologies due to the need to build testable and thus, hopefully, trustable OO systems. The standardization of semi-formal modeling methods, such as UML, reveals that testing can no longer be separated from specification/design/code stages: *design-for-testability* is a necessary basis for final-product reliability. 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 this paper, we present a model and a methodology for planning integration and non-regression testing from an OO model. We show how to produce a model of structural system test dependencies which evolves with the refinement process of the OO design. The model, that is the *test dependency graph*, serves as a basis for ordering classes and methods to be tested for non-regression and integration purposes (minimization of test stubs). The test dependency graph is a model which represents the main structural dependencies between components (classes or methods) in an OO system. Vertices of this graph represent the components and directed edges represent dependencies. In this paper, we present an algorithm which gives a strategy for ordering the tests of a system given its test dependency graph.

From a methodology point of view, we suggest a systematic separation between contractual and implementation aspects when modeling tests. It aims to provide a way of memorizing test sets and guide the reuse of contractual tests. This explicit separation is useful in a maintenance context, as well as for non-regression testing.

Section 2 opens on the definition of structural test dependencies which serve as a basis for defining the test dependency graph. The mapping from UML to the defined model is detailed as well as the test methodology. Section 3 concentrates on the test strategies (integration and non-regression) which are based on an original adaptation of Bourdoncle's algorithm to the testing problematics. Section 4 is devoted to a comparison of various integration strategies with the proposed optimized algorithm (a real-world case study illustrates this comparison). Section 5 presents and discusses related works.

II. Modeling structural test dependencies

II.1. Definitions

We need to introduce the concepts of *component*, *integration testing* and *stubs* which will be used in the rest of the paper.

Component: a basic test unit. In this paper, a component corresponds to a class, or to a specific method of a class in a refined design.

Integration testing: the way in which test is conducted to integrate components into the system. The integration is often conducted under incremental steps. One of the main difficulties for the safe integration of components is the minimization of the number of stubs to be written.

Stub: a dummy component used to simulate the behavior of a real component [Beizer90]. The test of a component X which calls a component Y, which is not already tested, implies the replacement of Y by a dummy component called *stub*. A *specific stub* is written if it simulates Y's behavior relatively to X use. The dummy component uses the same calling sequence but provides "canned" outcomes for its processing. A *realistic stub* is used if it simulates Y in every way. Realistic stubs can correspond to obsolete, but reliable, implementations of the stubbed component.

The testing effort will be related to the *number of stubs* that have to be written in an integration strategy. The number of stubs will be calculated depending on the types of stubs which can be written. If a *realistic stub* is used, then the number of stubs used for stubbing Y relatively to X and Z is one. If specific stubs are used, then two stubs are written for Y in order to test X and Z.

Non-regression testing: Non-regression testing is used when components of the systems evolve or when new components (and functionality) are added to the system. It aims at asserting both that changes are correct and that no regression bugs appear in the system due to the recent evolution. Generally, previous test sequences are launched to guarantee that the system has not regressed in terms of testing quality.

II.2. Test dependencies

In this section, we define test dependencies and the associated model called the Test Dependency Graph (TDG). A test dependency is mapped into the TDG as a directed edge between two nodes. Classes and/or methods from the system design are mapped into nodes in the TDG.

Let C_i and C_j be two components of a system S, the concept of system test dependency (an intuitively natural concept) is defined as follows:

Test dependency : A component class C_i is *test-dependent* from C_j if it uses some objects from C_j or inherits from C_j. This dependency relation is noted:

$$C_i \, R_{TD} \, C_j$$

We distinguish between implementation and contractual test dependencies. It brings a basis for reusing test sets that are independent from implementation choices and focus on an explicit separation of test sets into implementation and contractual ones.

Contractual Dependency (CD): A component C_i is *contractual-dependent* from C_j if it uses C_j or inherits from C_j, whatever the implementation choices are. We call a *Inheritance Contractual Dependency (ICD)*, a contractual dependency where C_i inherits from Cj and *a Client Contractual Dependency (CCD)* a contractual dependency such as C_i declares at least one object from Cj. Contractual dependency is noted $C_i \, R_{CD} \, C_j$ and is thus decomposed into inheritance dependency (noted $C_i \, R_{ICD} \, C_j$) or client one ($C_i \, R_{CCD} \, C_j$).

Implementation Dependency (ID): A component C_i is implementation-dependent from Cj if: $C_i \, R_{TD} \, C_j$ and not $C_i \, R_{CD} \, C_j$. It is noted $C_i \, R_{ID} \, C_j$

Test Dependency Graph (TDG): It is a directed graph whose nodes represent components (classes and included methods depending on the level of detail of the design) and whose directed edges represent test dependencies. In such a test dependency graph, loops may occur because components may be directly or indirectly test dependent from each other.

Types of dependencies:

Depending on the level of detail reached by the design, one may define test dependencies between classes, and if the level of detail is high, between methods of classes. We thus distinguish three types of test dependencies:

- *Class-to-class:* It is the first dependency that can be induced from a design. Each class is modeled by a node in a TDG: a directed edge exists between these nodes.
- *Method-to-class:* A method-to-class from *m* to *A* dependency may is induced if a method *m* has an object of a class *A* declared in its input signature. In the test dependency graph, a directed edge exists between the *m* node, modeling the method, and the *A* node, modeling the class. Polymorphism is modeled by having *m* transitively dependent on *A* subclasses through the inheritance dependency link.
- *Method-to-method:* Such dependency can be inferred only if details exist on the implementation body of a method. A method-to-method dependency from m_1 (of class *A*) to m_2 (of class *B*) exists if m_1 applies the method m_2 to an object of class *B*. In the Test Dependency Graph, a directed edge connects the m_1 node to the m_2 node and to all the redefinitions of m_2 in B subclasses (dynamic binding).

From a UML class diagram, only class-to-class and method-to-class test dependencies can be inferred into the TDG. Using information available in the UML dynamic diagrams would allow some method-to-method test dependencies to be also inferred If the code is available, then all the test dependencies are easy to extract for mapping into the model. Mapping a test dependency graph from the source code produces no method-to-class dependencies.

II.3. From UML to TDG

The TDG can be extracted from a design model such as a UML description of the OO system. The rules for creating a preliminary test dependency graph are given in Figure 1, 2 and 3. Figure 1 displays the basic mapping into class or method nodes, while Figures 2 and 3 outline respectively the mapping into method-to-method, method-to-class and class-to-class test dependencies.

The way of building a preliminary test dependency graph is illustrated in Figure 4. In this example, the model is simply deduced from a UML static class diagram. For sake of conciseness, rules for determining if a test dependency is an implementation or a contractual one are not detailed in this paper.

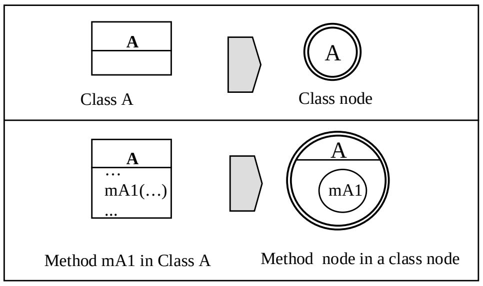

Fig. 1. UML to TDG: Nodes modeling

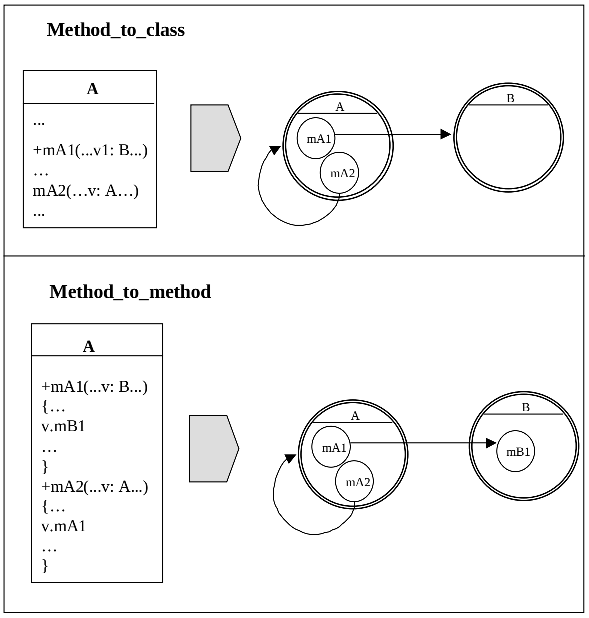

Fig. 2. UML to TDG:Types of dependencies

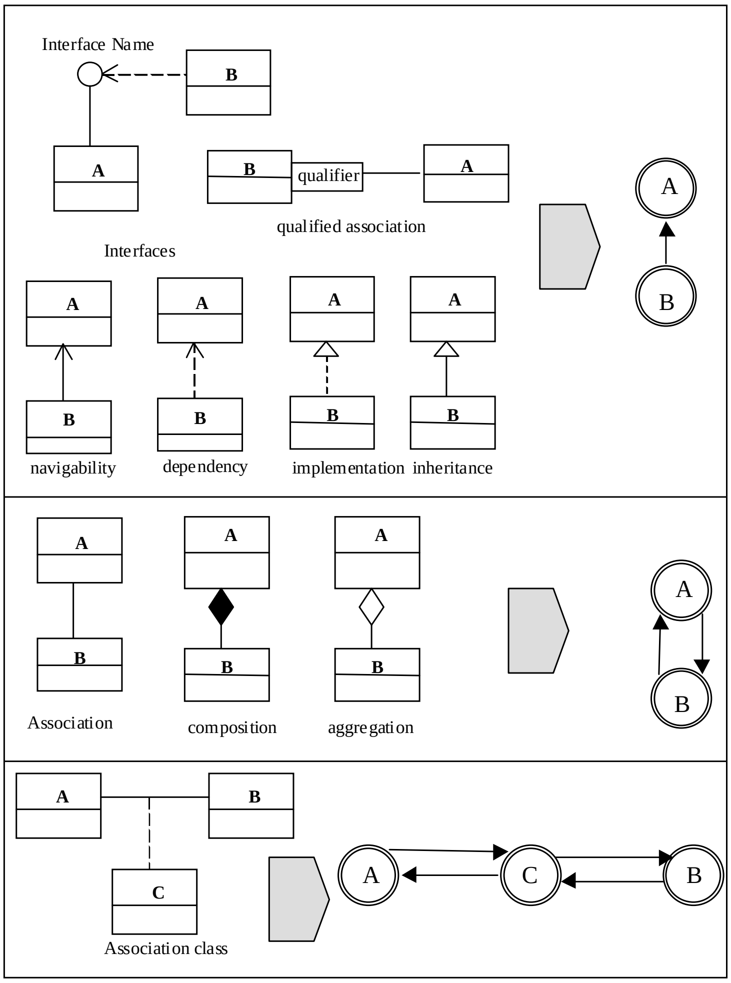

Fig. 3. UML to Test Dependency Graph: Class-to-class edges

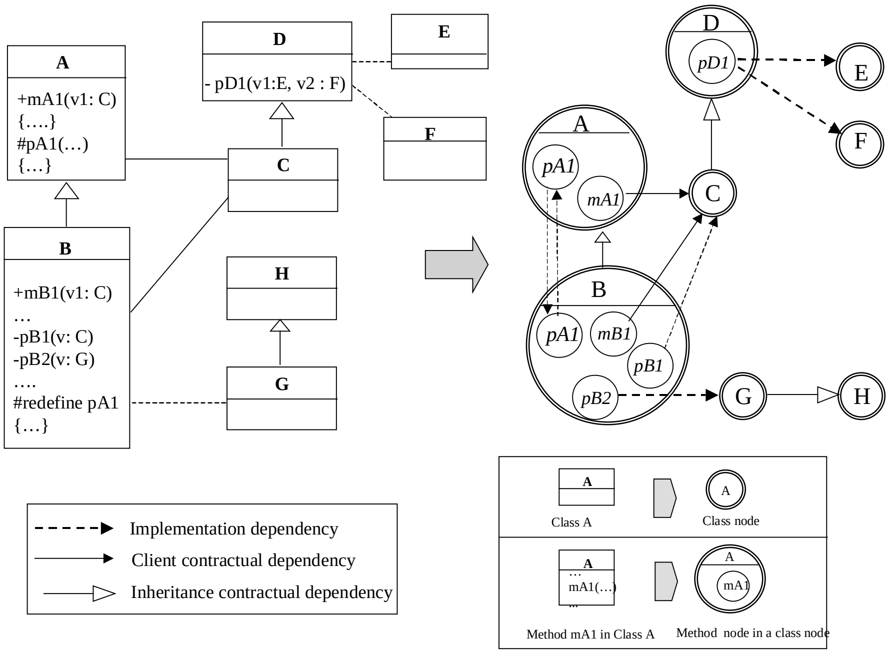

Fig. 4. A UML description and its associated preliminary test dependency graph

II.4. Towards a design-for-testability methodology

In this paper, we suggest a systematic separation between contractual and implementation aspects. It aims to provide a way of memorizing test sets and guide the reuse of contractual tests. This explicit separation is useful in a maintenance context as well as for non-regression testing. In a contractual client testing stage, only the contractual part of the graph will be used for planning test, being given a root component. So, depending on the level of testing, a contractual or an implementation dependent graph is produced. In the first case, the test plan will be reusable independently from the implementation choices while, in the second case, the test plan is specific to the implementation. Then, the selection context is defined, i.e. integration or non-regression testing context. In the first case, the test plan aims at minimizing the number of test stubs by ordering the components (classes or methods of classes) to be tested. In the second case, the test plan specifies the components to be tested after an evolution or a modification of a component (or a set of components) of the system

A pragmatic approach is well adapted to the present approach. Noting that the feasibility of class testing is similar to classical procedural testing, each component can be enhanced by the ability to invoke the unit test by itself: components are made *self-testable*. A self-testable component is a set containing a specification, a test suite and a given implementation. The production of self-testable components is presented in [Le Traon99] as well as a methodology for ensuring test quality. The approach is based on the following considerations:

- writing tests is easy at a unit class level [Beck98],
- verifying the test quality is feasible through fault injection. Mutation may also be used to select a reduced set of test cases. The process of estimating test quality can be automated,
- since a test driver consists in a set of self-test method calls, structural system tests are easy to launch.

Based on these considerations, the model of structural system test dependencies serves for guiding system self-test for integration and non-regression purposes. A preliminary test dependent graph is not a classical graph: graph algorithms cannot be directly applied on such structures due to the representation problem tackled by this preliminary modeling.

III. Test strategies based on hierarchical graph decomposition

The integration strategy is based on the decomposition of the test dependency graph. As a result, the components are ordered with respect to the minimization of *stubs*. The algorithm proceeds by decomposing the graph into its strongly connected component (existing loops are broken) and organizing the test by minimizing the stubs to be written.

III.1. Representation problem and graph normalization rules:

In the preliminary test dependency graph, two types of nodes exist for representing a class or a method. As a first modeling, we consider that a class node surrounds and includes all of its methods nodes. However, such preliminary modeling does not correspond to a classical graph representation, since three types of test dependencies exist. Edges may connect a class to another class or a method to another method, but also edges may connect a method node to a class node.

If the design is under refinement, all the possible types of test dependencies exist at the same time in the preliminary TDG. This modeling must be normalized into a classical graph representation to apply classical algorithms for testing. Two solutions are envisaged, with respect to the test meaning of graph edges (see Figure 5). Solution 1 consists in assimilating each method-to-method and method-to-class edge to a class-to-class edge. In this solution, most of the information from the design under refinement is lost for testing. To catch all the available information of the design under refinement, we propose the Solution 2. Methods nodes are separated from their class nodes, in which they were included in the first representation. From a test point of view, stubbing static attributes of a class is considered as a negligible effort: a method is not considered as being strongly test dependent from its class. So, we consider that each class is automatically test dependent from its included methods: to validate a class, its methods must be tested before.

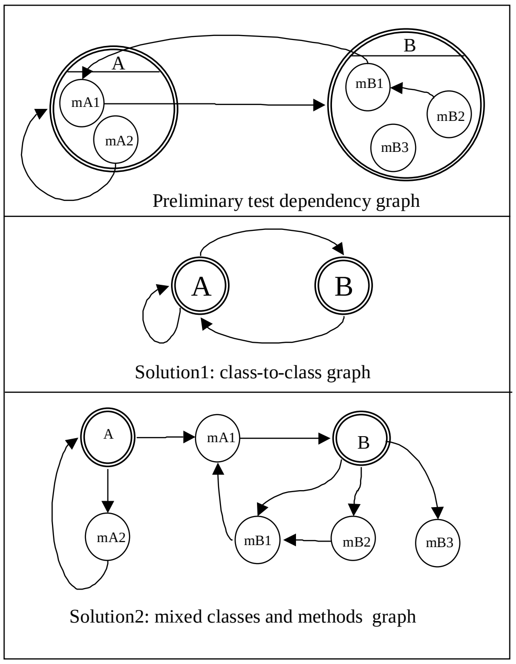

Fig. 5. The representation problem and two normalization rules

If the source code or a detailed model graph is available, then only method-to-method test dependencies will appear in the test dependency graph (for each method, the used methods are well known). In that last case, we can choose to apply a test strategy at a class level of detail or at a method level of detail. In the first case, all method-to-method dependencies are transformed into class-to-class dependencies (redundant edges are suppressed). This corresponds to the solution 1 (see Figure 5) normalization rule: the detailed information concerning method test dependencies is not lost for testing. In the latter, all information is used for planning testing. Class nodes are suppressed an only method nodes remain in the model after simplification. An integration test strategy can be specified which details in which order each method of each class has to be tested to minimize the testing effort.

If the design is poor, only class-to-class dependencies will appear, and consequently, the test dependency graph will be a classical graph.

Remark: The function which transform a preliminary test dependency graph into a test dependency graph through solution 2 is bijective. The demonstration is based on the fact that no class-to-method dependency exists in the preliminary TDG. The reverse function consists in including method nodes into their class nodes (detected by a class-to-method edge) and by deleting class-to-method edges.

III.2. Acyclic case

In the simplified case where the dependency graph is acyclic, it is clear that for the purpose of integration testing the natural strategy is to test components starting from decendants to ancestors in the graph. Such an order is given by a reverse topological ordering. Recall that a topological ordering of a directed acyclic graph (DAG) $G = (V,E)$ is an ordering $\leq$ of its vertices such that for every edge $v \to w$, $v \leq w$. Thus if we assume by this strategy that all vertices w such that $v \leq w$ are already tested and are supposed correct, v can be tested safely using its descendants w on which it depends. One can simply adapt a depth first search (DFS) of a DAG for printing the vertices in the reverse topological ordering. This ordering is directly applicable for the testing strategy.

III.3. General case

However, in the general case the test dependency graph has cycles. The strategy thus has to take them into account. This is done using a decomposition of the graph into its strongly connected components (SCCs) and breaking SCCs which implies the development of stubs.

Strongly connected components

Recall that a strongly connected component of a graph is a subgraph such that for any pair if vertices v and w, there is a path from v to w and vice versa. Consider that two vertices v and w are equivalent if they are in the same strongly connected components.

The quotient of the graph by the equivalence relation defines a DAG, the reduced DAG of strongly connected components where each vertex of the DAG is an SCC. Tarjan's algorithm [Tarjan72] computes this DAG and prints SCCs in the reverse topological ordering. We do not want to detail the algorithm here, but simply give some characteristics useful for our purpose.

Tarjan's algorithm

The algorithm is based on a DFS with a supplementary stack which stores vertices which SCC is not completed. Vertices are numbered according to their first visit by the DFS. We note *v.num* this attribute. The DFS defines a partition of the edges into four classes (see Figure 6):

- tree edges lead from a vertice to an unvisited vertice. Tree arcs define a spanning forest of the graph.
- forward edges are non tree edges which go from a vertice to a descendant. These edges play no role in the computation of SCCs.
- fronds go from a vertice to an ancestor in the spanning forest,
- cross edges are the remaining edges. They go from a vertice to a different subtree in the forest.

The *root* of an SCCs is the first vertex of the SCC which is traversed during the DFS. The principle of the algorithm is to detect roots of SCCs. For that purpose it uses an attribute *lowlink* to vertices *v*which is the minimum number *w.num* of vertices *w* which are accessible from *v* by tree edges and at most one cross or frond edge. A root *r* has the property that *r.lowlink* = *r.number*. The algorithm has linear complexity in time and space in the size of the graph.

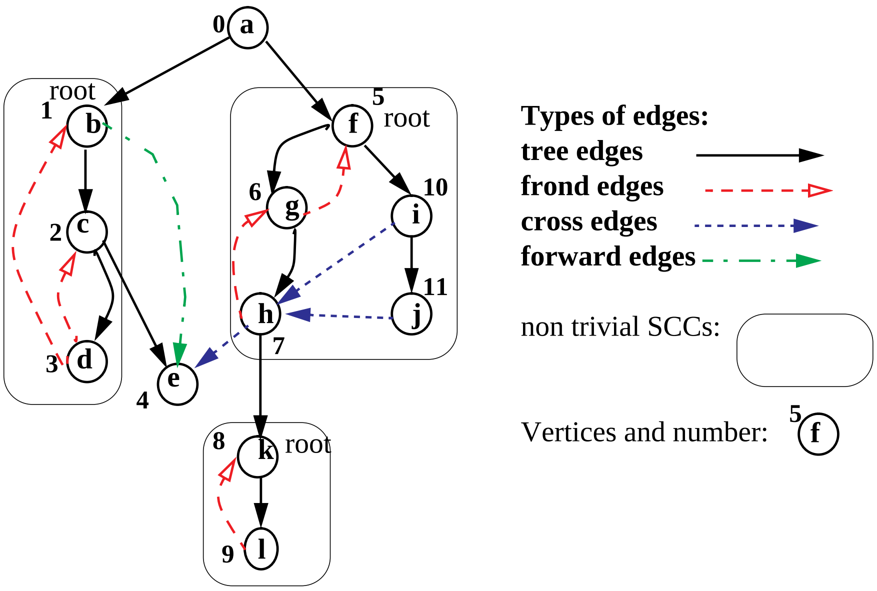

Fig. 6. Tarjan's algorithm and partitioning of edges

As the reduced DAG of SCCs is a DAG, we can use the same scheme as the strategy proposed in the acyclic case. The inverse of the topological ordering given by Tarjan's algorithm gives an ordering in which components in SCCs can be tested for integration.

Bourdoncle's algorithm

Bourdoncle's algorithm is an adaptation of Tarjan's algorithm which was originally designed for static analyses purpose and which can help in finding these stubs and to minimize their number. The main idea is to apply recursively Tarjan's algorithm to each non trivial SCC, starting from its root, after having removed all fronds which enter the root. Removing these edges break some cycles in an SCC but not necessarily all of them. It is why Tarjan's algorithm has to be applied recursively until all cycles are broken. This algorithm has complexity quadratic in the size of the graph. In the example of figure 7 the first call of Tarjan decomposes the graph into five SCCs: $\{\underline{a}\},\{\underline{b},c,d\},\{\underline{e}\},\{\underline{f},g,h,i,j\},\{\underline{k},l\}$ with roots underlined. Trivial SCCs $\{\underline{a}\}$ and $\{\underline{e}\}$ have no cycle thus nothing has to be done. For non trivial SCCs, fronds entering the root are removed. For example $d \rightarrow b$ is removed from $\{\underline{b},c,d\}$. The recursive call to Tarjan's algorithm then gives two SCCs: $\{\underline{b}\}$and $\{\underline{c},d\}$. Again frond $d \rightarrow c$ is removed from the only non trivial component and the last call gives two trivial SCCs $\{\underline{c}\}$and $\{\underline{d}\}$. The same decomposition is applied to $\{\underline{f},g,h,i,j\}$and $\{\underline{k},l\}$.

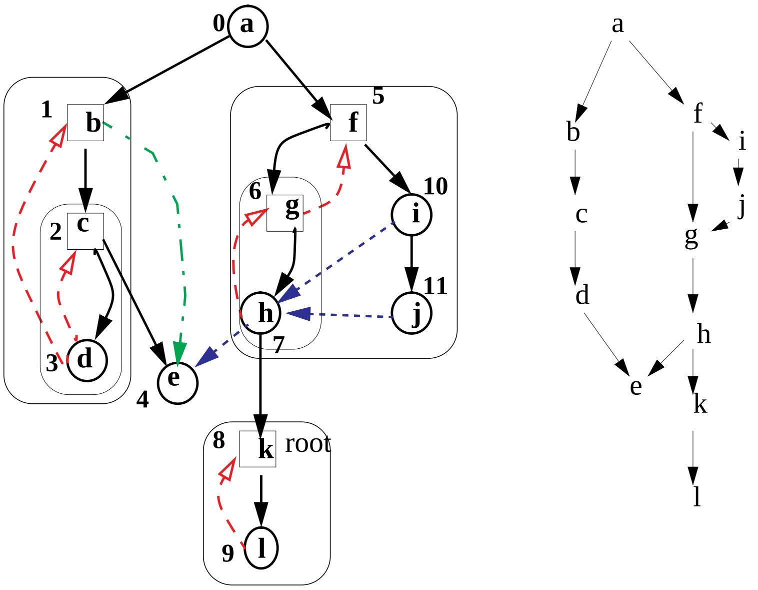

Fig. 7. *Ordering given by Bourdoncle's algorithm*

The decomposition is not unique as it depends on the order in which successors of a vertex are explored. A possible decomposition computed by the algorithm is given by the left part of Figure 7. Roots of SCCs in recursive calls are identified by squares and SCCs are surrounded. The partial ordering defined by this decomposition is given by the right part of Figure 7.

Now let us interpret this in terms of testing strategy. As the decomposition is a partial ordering, this gives several possibilities. For example, we can start by testing the component *l*. As *l* is the source of a frond to a root, it needs as a stub the target of the frond, namely *k*. When *l* has been tested using *stub*(*k*), *k* can be tested as it depends on *l*. Now *e* has to be tested. It could have been tested before *l* and *k*. Then we can proceed by the left or right branch in any order. For the test of *d*, stub(b) and stub(c) are needed as $d \to b$ and $d \to c$ are fronds to roots. Then *c* can be tested using *d* and *e* and then *b* using *c* and *e*. On the right branch, *h* is tested first using stub(g), *k* and *e*. Then *g* is tested using *h* and stub(f). *j* is tested using *h* and then *i* using *j* and *h*and finally *f* using *g* and *i*. Finally *a* is tested using *b* and *f*.

III.4. Improved integration strategy

Choosing the root of each SCC in recursive calls as a stub insures that cycles are broken. Nevertheless, as the objective of the original algorithm of Bourdoncle's was different from our, the strategy is not ideal for the purpose of minimizing stubs. We propose a slightly different strategy where a stub should break as much cycles as possible. Finding the optimal one in an SCC is exponential while taking the root is of constant complexity. So in order to keep the same complexity as in Bourdoncle's algorithm, we propose a choice of stubs which adds no cost to the algorithm. The idea is to choose a vertex with maximal number of incoming or outcoming fronds where a frond in a DFS is an edge which comes from a descendant to an ancestor.

The first call to Tarjan's algorithm is identical to Bourdoncle's algorithm except that a counter is associated to each vertex. When a frond is detected, the counters of source and target vertices are incremented. When a root of SCC is detected, all vertices of the SCC are popped from the SCC stack and the vertex with maximal counter is selected. Recursive calls are applied to non-trivial SCCs from the selected vertices, removing incoming edges to this vertices and follow the same scheme.

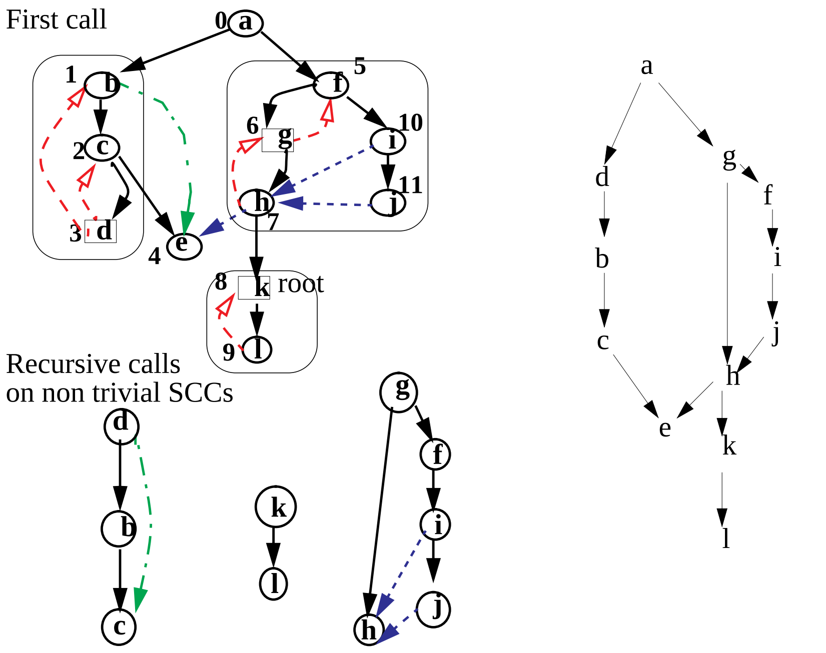

Fig.8.: *Optimized algorithm and corresponding ordering*

The application of the algorithm on the example of figure 6 is detailed in figure 8. The first call identifies the three non-trivial SCCs. In SCC {*b*,*c*,*d*}the selected vertex is *d*. In the recursive call, *c*→*d* is deleted and the remaining subgraph is acyclic. In {*f*,*g*,*h*,*i*,*j*}, the vertice *g* is selected, edges *f*→*g* and *h*→*g* are deleted and the remaining graph is acyclic. In {*k*,*l*}, *k* is deleted and the remaining subgraph is acyclic. This gives the ordering in the right part of the figure.

In terms of testing this gives the following strategy. *l* is tested using stub(k) and then *k* is tested. *e* is tested. Now for the SCC {*b*,*c*,*d*}, *c* is tested using stub(d) and *e*, then *b* is tested using *c* and *e* and finally *d* is tested using *c*. For{*f*,*g*,*h*,*i*,*j*} , *h* is tested first using stub(g), *e* and *k*. Then *j* is tested using *h*, and *i* using *h*, followed by *f* tested using stub(g) and *i* and finally *g* is tested using *h* and *f*. Now *a* can be tested using *b* and *f*.

Compared with Bourdoncle's algorithm, this algorithm gives better results on this example. In terms of cost of the algorithm, we can see that we only have two levels of recursive calls but three for Bourdoncle's algorithm. This is because the strategy selects vertices which breaks more cycles. In terms of effort for stubs, it is also better. We only have three *realistic stubs* (k, d, g) or four *specific stubs* corresponding to edges *l*→*k*, *c*→*d* , *f*→*g* and *h*→*g*. Bourdoncle's algorithm gives five *realistic stubs* (k, b, c, f, g) or five *specific stubs* corresponding to edges *l*→*k*, *d*→*c* , *d*→*b*, *h*→*g* and *g*→*f* .

III.5. Non-regression strategy

For non-regression, the previous optimized strategy can be used to order components to be re-tested when a component (or a set of components) is modified. The algorithm has to be applied on the SCC in which the modified component is included and on its predecessors connected parts. However, a criterion is used to qualify the parts of the system which are re-tested, i.e. coverage of each dependency path or from only direct dependencies. Moreover, when planning non-regression testing, contractual/implementation aspects should be taken into account for deciding whether only contractual (client) dependencies are re-tested or implementation ones. If application is not safety-critical, a weak non-regression strategy can be applied (contractual aspects and direct dependencies). This leads to the following coverage criteria (in the sense of Weyuker criteria [Rapp85]) which may guide the application of the algorithm for reducing the testing effort.

Non-regression test adequacy criteria: For testing a component C into a system, two basic criteria may be used. The weakest consists of testing components which are directly dependent from C (direct-dependencies coverage). The strongest consists of testing each component which is included into a path containing C (all-dependencies coverage). The notion of direct and all dependency adequacy criteria may be applied to each type of dependency (contractual inheritance, contractual client and implementation dependencies). A simple partial order relationship exists between adequacy criteria as presented for contractual dependencies in Figure 7 (the stronger criterion is on the top of the figure).

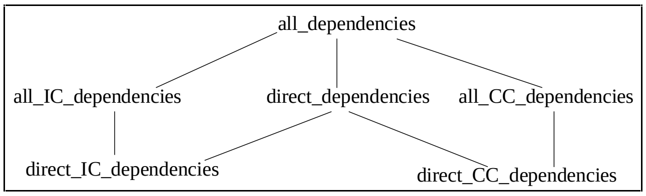

Fig. 9. Partial order between test adequacy criteria

IV. Case study and results

To make this paper more concrete, we will use a case study from the telecommunications domain. 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). The detailed description of an SMDS server design and implementation can be found in [Jézéquel96]. Its UML class diagram is presented Figure 10. Each class has a number which corresponds to a node in the TDG (Figure 11). It is composed only by class-to-class test contractual test dependency.

Our purpose is to produce the test plan for integration testing and to compare the stubs gain with various strategies. The stub counting is performed both with specific to a use stubs (specific stub) and realistic ones. It provides a first estimate of the integration testing effort.

The strategies are the following. They have been applied directly and after a first decomposition of the TDG into connected parts using Tarjan's algorithm.

Random component selection (RC): The components are tested in a uniform random order.

Most used component selection (MC): The components are tested from the most used (maximum of predecessors) to the less one. Each tested component is suppressed from the graph before choosing another one. This strategy is deterministic (the first candidate is chosen at each step).

Random thread of dependency selection (RT): At each step of testing, a component is randomly chosen for testing. Then, one of its predecessor is randomly chosen and so on along a test dependency thread until there are no more non-tested predecessors. Steps are repeated until all components are tested.

Most used thread of dependency selection (MT): Similar as RT except for the component selection mode for each step beginning. The chosen component is a component with a maximum of predecessors.

RT and MT selection modes correspond to an intuitive integration strategy. A component is tested which uses an already used component. A thread of test dependency is thus followed and gives the test order. MC corresponds to a systematic choice of a good candidate to be stubbed at a given step while RC is a pure random selection.

The complete result is represented by a decision tree to decide parallel testing steps. For sake of conciseness, we do not present the complete tree but only one sequential solution. A solution produced by an application of the optimized algorithm is the following:

(2, 29, 37, 34, 17, 12, 16, 18, 15, 27, 25, 24, 19, 21, 20, 13, 14, 7, 1, 6, 33, 22, 10, 31, 32, 3, 28, 30, 26, 8, 9, 36, 23, 11, 4, 5, 35)

In terms of realistic stubs, nine stubs are created: 10, 13, 15, 22, 24, 26, 28, 30, 32. With the optimized solution, 20 specific stubs have to be created or 9 realistic ones (or a mixed solution). Bourdoncle's algorithm gives the following result: 23 specific stubs or 12 realistic ones. The chosen realistic stubs are the following: 10, 13, 15, 14, 18, 20, 22, 24, 26, 28, 30, 32.

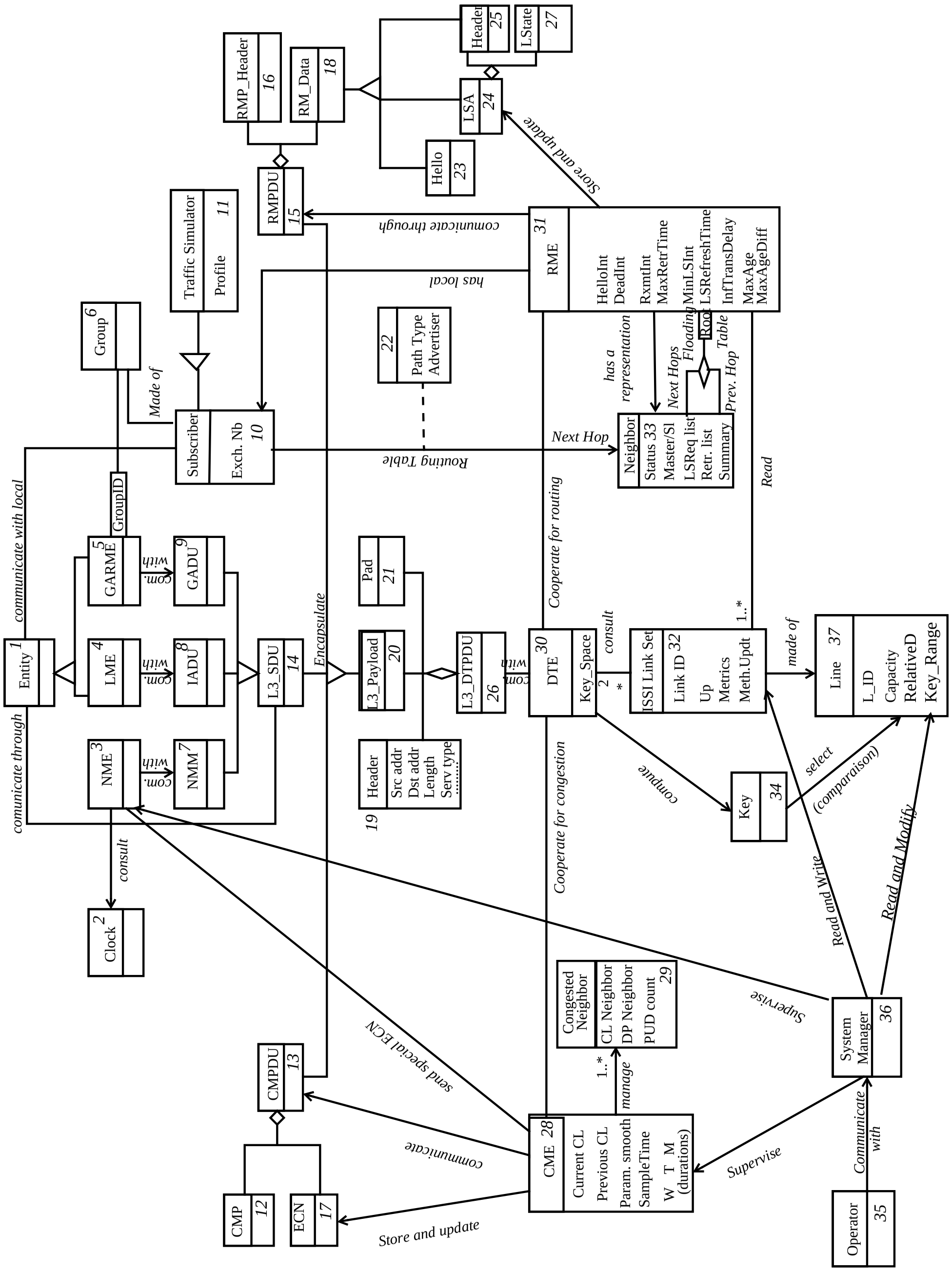

Fig. 10. SMDS UML class diagram

The minimum, mean and maximum number of either specific and realistic are given in Figure 12 for specific stubs and in Figure13 for realistic ones. Random strategies have been applied 100,000 times for the SMDS example. One surprising result is the fact that the RT selection criterion gives the worse selection choice. Only the proposed optimized strategy reaches the lowest score. The gain in terms of number of stubs is more important with realistic stubs integration testing. One interest of the optimized strategy consists in locating the best candidate components for creating realistic stubs. On such an example, 37 components are tested. Concerning random based selection, the random criteria must choose into a set of n! possible ordering of the n components (37! in this example). In a bigger example, the probability to select a good ordering falls and the gap increases between optimized strategy and random ones. Concerning a deterministic selection, such as MC, results show that the optimized strategy gives better results in practice. In fact, the experience shows that the optimized strategy is an efficient heuristic that results in an ordering close to the optimum solution.

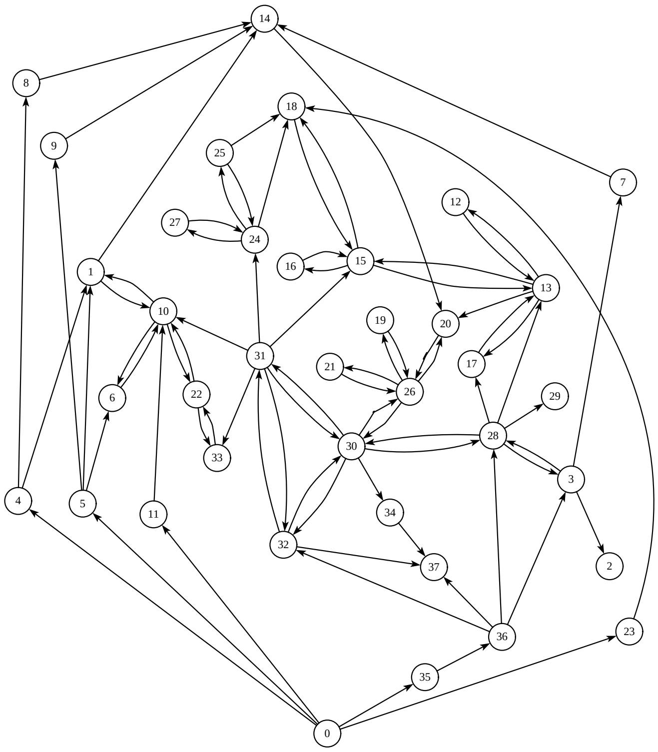

Fig. 11. Test Dependency Graph of the SMDS case study

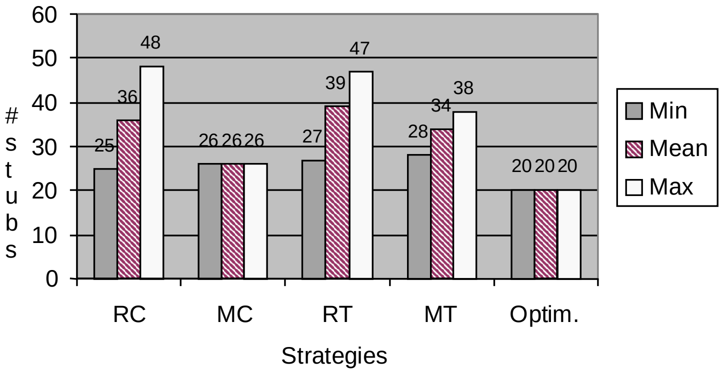

Fig. 12. Specific stubs counting

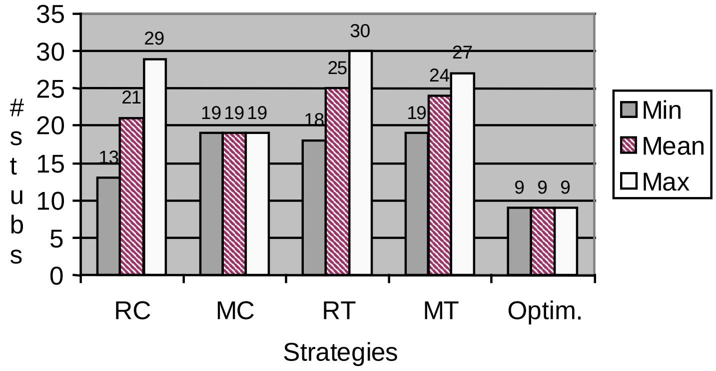

Fig. 13. Realistic stubs counting

V. Related works

Very few of the numerous first-generation books on analysis, design, and implementation of object-oriented software explicitly address V&V issues. Despite this initial lack of interest, testing of object-oriented systems is receiving much more attention (see [Binder96] for a detailed state of the art).

Concerning OO testing techniques, most of the works focus on the dynamic aspects of OO systems: a system is viewed as a set of cooperating agents, modeling objects, and modeled with FSM, or equivalent object-state modeling [Binder94, Jorgensen94, Kung96]. Such works have to deal with limitations concerning computational expense of mapping objects behaviors into the underlying model. One solution consists in decomposing the program into hierarchical and functionally coherent parts. In such approaches, this decomposition provides a framework for unit, integration and system test definition. In [McGregor94], the waterfall model is overtaken and an integrated test and development approach is proposed. These state-based models constrain the design methodology to divide the system into small parts with respect to behavioral complexity. Concerning test strategies, our work is very much along the lines of [Harrold92, Kung96] approaches. In particular, Kung proposes a method for identifying affected classes during maintenance and giving a desirable order to test these classes. The used object relation graph model serves for ordering classes to be tested for non-regression, integration and maintenance purposes. In this paper, we differ by the way of organizing test and by the underlying test methodology. In terms of test organization, the test attributes which are modeled and the algorithms used for optimizing integration are original and compared to various other strategies. In terms of test methodology, the approach is adapted to early test planning (e.g. from a UML model) and is more particularly suited to self-testable unit components. Self-testable components have the ability to launch their own unit tests as detailed in [Le Traon99]. The test plan evolves during all design refinement steps (including code production).

VI. Conclusion

In this paper we have presented a graph model called Test Dependency Graph for the representation of test dependencies in OO systems. This model distinguishes the type of dependencies: method to method, class to class and method to class. We have seen how this model can be extracted from an UML description. From this general graph model, we have detailed the extraction a simplified model by the externalization of methods nodes from classes (graph normalization). This representation is then suited for graph algorithms.

A graph algorithm which computes a strategy for integration testing with a complexity quadratic in the size of the model has been detailed The algorithm provides an efficient testing order for minimizing the number of stubs. The algorithm has been illustrated on a real world case study and compared with other strategies. The results of the experiment seem to give nearly optimal stubs with a low cost despite the exponential complexity of getting optimal stubs.

References

[Beck98] K. Beck and E. Gamma, "Test-Infected: Programmers Love Writing Tests,"Java Report, July 1998, 37-50.

[Beizer90] B. Beizer, “Software testing techniques,” Van Norstrand Reinhold, 1990. ISBN 0-442-20672-0.

[Binder94] Robert V. Binder, "Design for Testability with Object-Oriented Systems," Communications of the ACM, v 37, n 9, September 1994, 87-101.

[Binder96] Robert V. Binder. Testing object-oriented software : A survey. Journal of Sofware Testing, Verification and Reliability, 6(125-252), 1996.

[Bourdoncle93] 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.

[Harrold92] 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, Calif. 68-80.

[Jézéquel96] Jean-Marc Jézéquel, “Object Oriented Software Engineering with Eiffel,” Addison-Wesley, mar 1996. ISBN 1-201-63381-7.

[Jorgensen94] Paul C. Jorgensen and Carl Erickson, "Object-Oriented Integration Testing," Communications of the ACM, v 37, n 9, September 1994, 30-38.

[Kung96] David C. Kung, Gao, Jerry, Chen, Cris., "On Regression Testing of Object-Oriented Programs," The Journal of Systems and Software. Jan 1996 v 32 n 1.

[LeTraon99] Yves Le Traon, Daniel deveaux and Jean-Marc Jézéquel, “Self-testable components: from pragmatic tests to a design-for-testability methodology,” In proc. of TOOLS-Europe’99. TOOLS, June 1999. To be published.

[McGregor94] 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.

[Rapps85] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Transactions on Software Engineering, vol. 11, pp. 367-375, 1985.

[Tarjan72] R. Tarjan, " Depth-first search and linear graph algorithms", SIAM J. Comput., vol.1, n 2, June 1972, 146-160.