

Efficient Object-Oriented Integration and Regression Testing

Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, and Pierre Morel

Abstract—This paper presents a model, a strategy, and a methodology for planning integration and regression testing from an object-oriented model. It shows how to produce a model of structural system test dependencies which evolves with the refinement process of the object-oriented design. The model (test dependency graph) serves as a basis for ordering classes and methods to be tested for regression and integration purposes (minimization of test stubs). The mapping from Unified Modeling Language to the defined model is detailed as well as the test methodology. While the complexity of optimal stub minimization is exponential with the size of the model, an algorithm is given that:

- computes a strategy for integration testing with a quadratic complexity in the worst case,
- provides an efficient testing order for minimizing the number of stubs.

Various integration strategies are compared with the optimized algorithm (a real-world case study illustrates this comparison).

The results of the experiment seem to give nearly optimal stubs with a low cost despite the exponential complexity of getting optimal stubs. As being a part of a design-for-testability approach, the presented methodology also leads to the early repartition of testing resources during system integration for reducing integration duration.

Index Terms—Graph algorithm, integration testing, object-oriented modeling, regression testing, software testing.

I. INTRODUCTION

Acronyms¹

OO	object oriented
UML	Unified Modeling Language [1]
SIT	Strategy for Integration Testing
CD	contractual dependency
CCD	client CD
DAG	directed acyclic graph
DFS	depth-first search
ICD	inheritance CD
ID	implementation dependency
SCC	strongly connected components
SMDS	switched multi-megabits data service
TD	test dependency
ITD	implementation TD
CTD	contractual TD

Manuscript received August 31, 1999; revised October 1, 1999.

The authors are with IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France (e-mail: yletraon, Thierry.Jeron, Jean-Marc.Jezequel, Pierre.Morel@irisa.fr).

Responsible editor: M.A. Vouk.

Publisher Item Identifier S 0018-9529(00)06199-6.

¹The singular and plural of an acronym are always spelled the same.

TDG TD Graph (our model in this paper)

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.

This paper presents a model, TDG, and a methodology for planning integration and regression testing from an OO model. It shows how to produce a model of structural system TD which evolves with the refinement process of the OO design. TDG is a basis for ordering classes and methods to be tested for regression and integration purposes (minimization of test stubs). TDG is a model which represents the main structural dependencies between components (classes or methods) in an OO system. Vertexes of this graph represent the components; directed edges represent dependencies. This paper presents SIT which gives a strategy for ordering the tests of a system, given its TDG.

From a methodology point of view, a systematic separation between contractual and implementation aspects is suggested when modeling tests. It aims to

- provide a way of memorizing test sets,
- guide the reuse of contractual tests.

This explicit separation is useful in a maintenance context, as well as for regression testing.

Section II defines structural TD which serve as a basis for defining the TDG. The mapping from UML to the defined model is detailed as well as the test methodology.

Section III concentrates on the test strategies (integration and regression) which are based on an original adaptation of the Bourdoncle algorithm to the testing problematics.

Section IV compares various integration strategies with the optimized SIT; a real-world case study illustrates this comparison.

Section V presents and discusses related works.

II. MODELING STRUCTURAL TD

A. Definitions

The concepts of component, integration testing, and stubs are introduced; they are used in the rest of the paper.

- **Component:** a basic test unit; 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 often uses incremental steps. One of the main difficulties for the cost-efficient 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 [2]. The test of a component X that calls a component Y, that 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 is related to the number of stubs that have to be written in an integration strategy. The number of stubs is 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 relative to X and Z is 1. If specific stubs are used, then 2 stubs are written for Y in order to test X and Z.
- Regression testing: this is used when components of the systems evolve or when new components (and functionalities) are added to the system. It tries to assert 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.

B. Test Dependency

Notation:

S	the system
C_i, C_j	two components of S
R_{TD}	relation of TD
R_{CD}	CD relation
R_{ID}	ID relation
R_{CCD}	CCD relation

1) *General Definitions:* TD and the associated model, TDG, are defined. A TD 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.

- TD: C_i is TD on C_j if it uses some objects from C_j or inherits from C_j . This dependency relation is:

$$C_i R_{TD} C_j;$$

and it means that we cannot really test C_i unless C_j is considered correct (e.g., tested).

Test cases can cover:

- private/protected parts of the system where tests are linked to private implementation choices that can be changed;
- public/interface parts of the system where tests concern the contractual aspects of a class.

We thus distinguish between ID and CD.

ITD and CTD are different; they

- provide a basis for reusing test sets that are independent of implementation choices,

- focus on an explicit separation of test sets into implementation and contractual ones.

While most system modifications concern implementation changes or new functionality addition, ID are less stable than CD for system modifications. Thus, test sets based on CD are reusable when ITD are dedicated and fixed to a given version.

- CD: A component C_i is CD from C_j if it uses C_j or inherits from C_j , whatever the implementation choices are.
- ICD is a CD where C_i inherits from C_j ; it is shown as: $C_i R_{ICD} C_j$
- CCD is a CD where C_i uses at least 1 object from C_j ; it is shown as: $C_i R_{CCD} C_j$.
- ID: A component C_i is ID from C_j if: $C_i R_{TD} C_j$, and not $C_i R_{CD} C_j$; it is written as $C_i R_{ID} C_j$.
- TDG 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 TD. In such a TDG, loops can occur because components can be directly or indirectly TD on each other. This is quite usual in OO systems designed "by the book," because of the reliance on dynamic dispatch and call back to implement variability in many design patterns.

2) *Types of Dependencies:* Depending on the level of detail reached by the design, one may define TD between classes, and if the level of detail is high, between methods of classes. We thus distinguish three types of TD:

- Class-to-class: It is the first TD 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 TD is induced if a method m has an object of a class A declared in its signature. In the TDG, 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 TD 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 TDG, 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 TD can be inferred into the TDG. Using information available in the UML dynamic diagrams would also allow some method-to-method TD to be inferred. If the code is available, then all the TD are easy to extract for mapping into the model. Mapping a TDG from the source code produces no method-to-class TD. Indeed, method-to-class TD basically abstract method-to-method TD.

C. 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

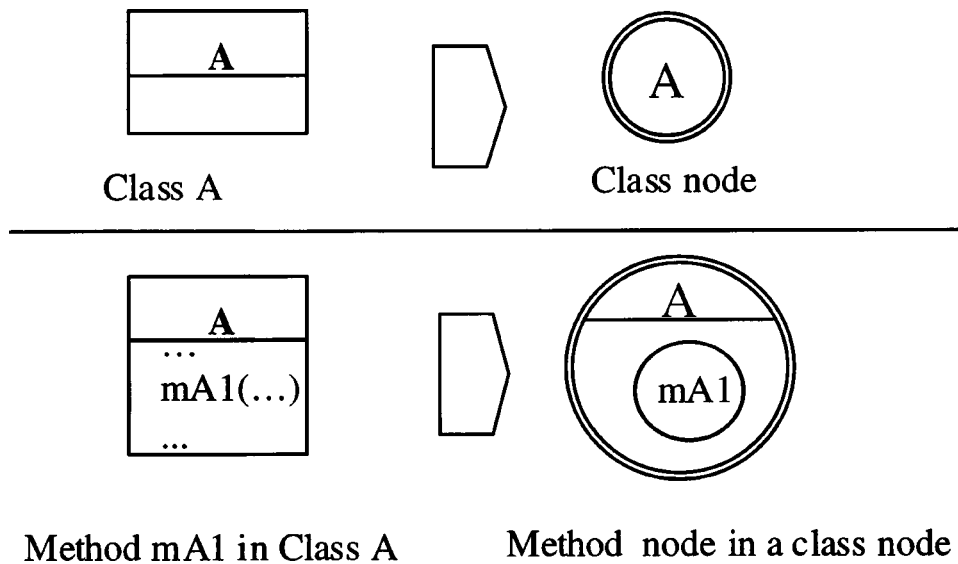


Fig. 1. UML to TDG: Nodes Modeling

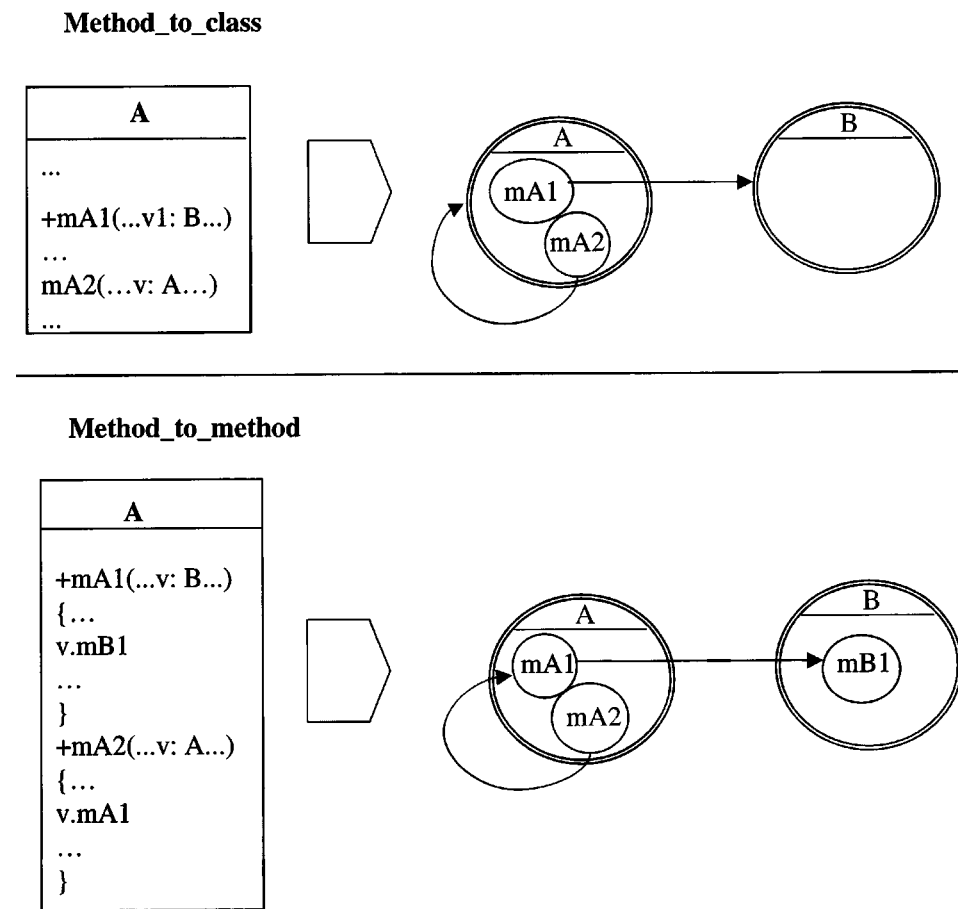


Fig. 2. UML to TDG: Types of Dependencies

a preliminary TDG are given in Figures 1–3. Fig. 1 displays the basic mapping into class or method nodes. Figs. 2 and 3 outline the mapping into method-to-method, method-to-class, and class-to-class TD.

Remark: At a method level of detail, if an explicit method-to-class TD between a method m (from a class A) and a class B is mapped on the model, then the class-to-class TD between A and B becomes redundant and is suppressed from the model.

The following 4 rules determine if TD are CD or ID:

- An ICD from B to A is created if B inherits from A .
- If a protected method m from A is redefined in B , a bi-directed TD exists between these methods (due to polymorphism interdependency).
- A method-to-class ID from a method m_1 to a class C or method m_2 is created if m_1 is private or protected and uses m_2 or inherits from C .

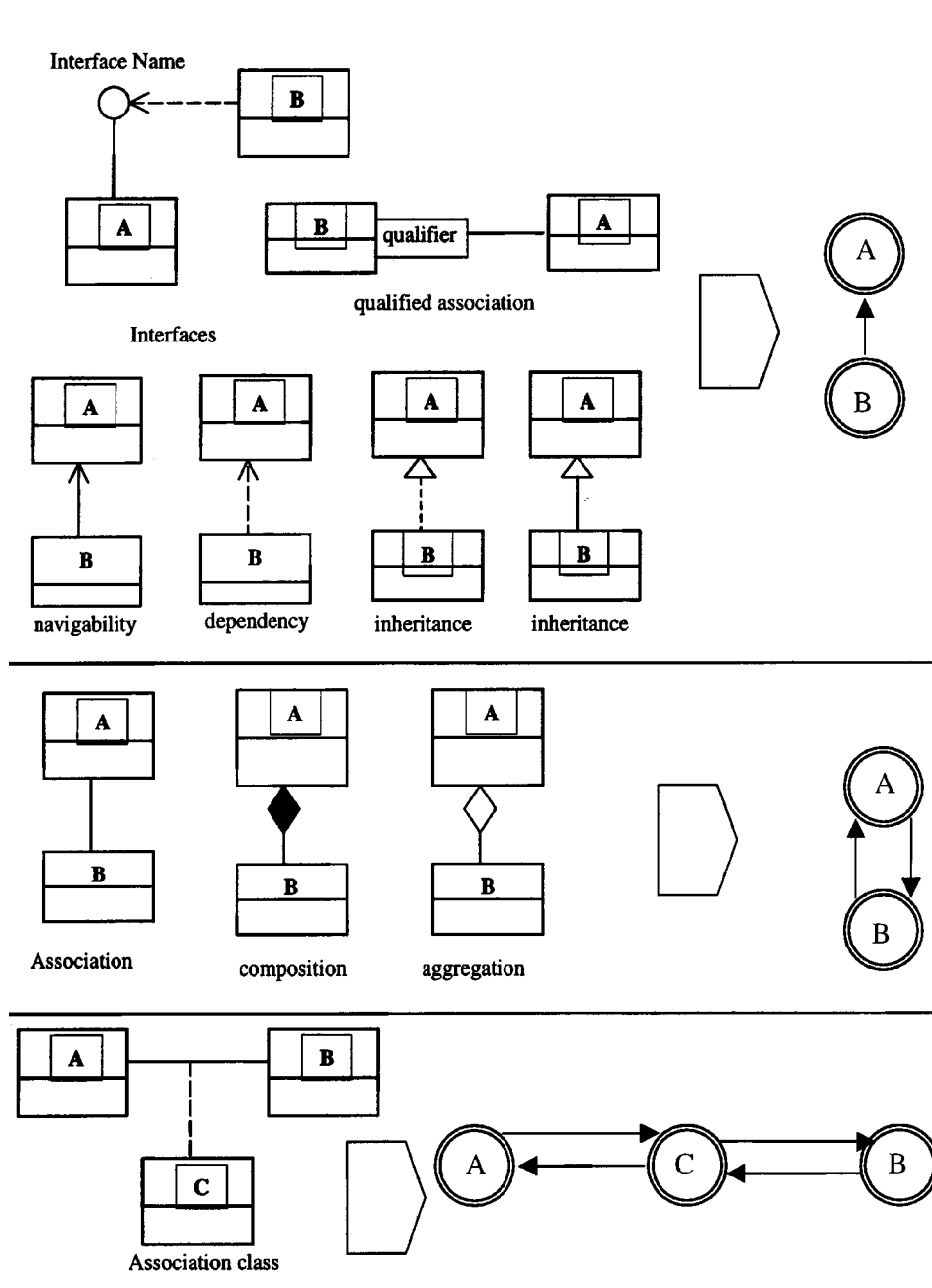


Fig. 3. UML to TDG: Class-to-Class Edges

- A method-to-method ID is created between a method m_1 and a method m_2 if the body of m_1 uses m_2 . This value is not available from a UML description, because no action language is associated with UML. Recent work suggests the addition of such language to UML modeling [3]. Nevertheless, from a source code, method-to-method dependencies can be determined.

Fig. 4 illustrates “building a preliminary TDG.” In this example:

- the model is simply deduced from a UML static class diagram;
- the redundant class-to-class TD have been suppressed when method-to-class TD are defined;

- a bi-directed method-to-method between pA1 of class A and pA1 of class B has been defined because of the possible polymorphic use of pA1 (for example due to the fact that mA1 could use pA1).

D. Towards a Design-for-Testability Methodology

This paper suggests a systematic separation between contractual and implementation aspects. It aims to provide a way of memorizing test sets and to guide the reuse of contractual tests. This explicit separation is useful in a maintenance context as well as for regression testing. In a contractual client testing stage, only the contractual part of the graph is used for planning test, being given a root component. So, depending on the level of testing, a CD or ID graph is produced. For a CD graph,

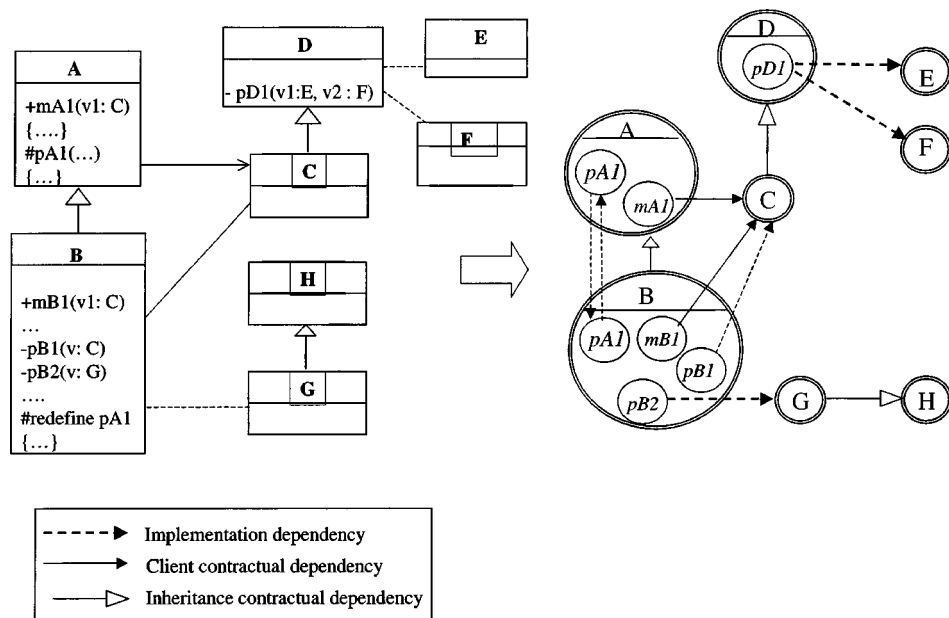


Fig. 4. A UML Description and Its Associated Preliminary TDG

the test plan is reusable independently from the implementation choices while, for an ID graph, the test plan is specific to the implementation. Then, the selection context is defined: integration or regression testing context. For integration testing, the test plan aims at minimizing the number of test stubs by ordering the components (classes or methods of classes) to be tested. For regression testing, 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 preliminary TDG 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 decomposition of the TDG. 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.

A. Graph Normalization Rules

In the preliminary TDG, there are 2 types of nodes for representing a class or a method. As modeling step #1, 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 TD exist. Edges can connect:

- a class to another class, or
- a method to another method, or
- a method node to a class node.

If the design is under refinement, then all the possible types of TD 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 possible, with respect to the test meaning of graph edges (see Fig. 5).

- Solution 1 assimilates each method-to-method and method-to-class edge to a class-to-class edge. 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 use solution 2.
- Solution 2 separates methods nodes 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 TD from its class. So, we consider that each class is automatically TD from its included methods: to validate a class, its methods must be tested before the class.

1) *The Source Code or a Detailed Model Graph is Available*: Only method-to-method TD appear in the TDG (for each method, the used methods are well known). We can apply a test strategy at a

- class level of detail, or
- method level of detail.

At the class-level, all method-to-method dependencies are transformed into class-to-class dependencies (redundant edges are suppressed). This corresponds to solution 1 (see Fig. 5) normalization rule: the detailed information concerning method TD is lost for testing. At the method level, all information is used for planning testing. Class nodes are suppressed, and 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.

2) *The Design Is Poor*: Only class-to-class dependencies appear; consequently, the TDG is a classical graph.

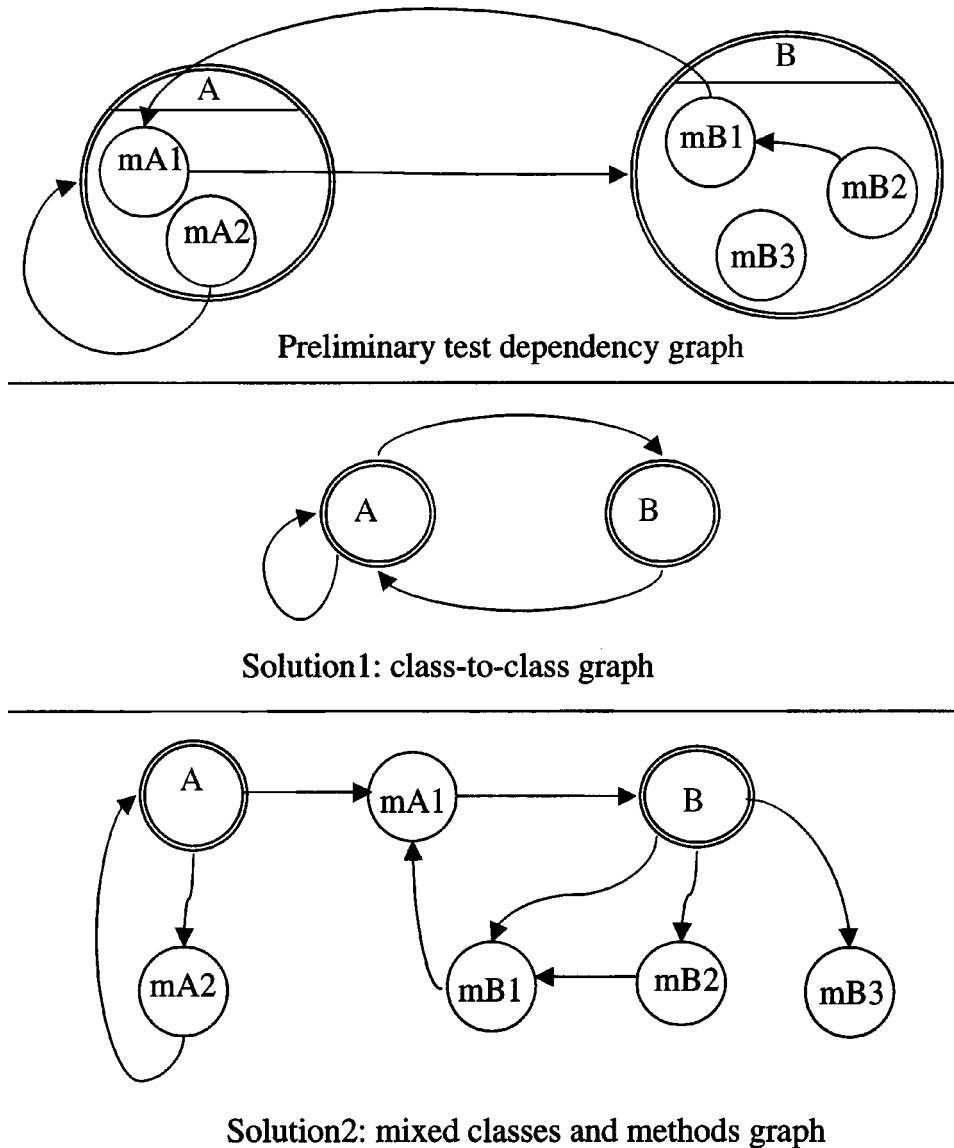


Fig. 5. Representation Problem and 2 Normalization Rules

3) *Remark:* The function which transforms a preliminary-TDP into a TDP through solution 2 (Fig. 5) 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.

B. Acyclic Case

In the simplified case where the dependency graph is acyclic, it is clear that, for the purpose of integration testing, the reasonable strategy is to test components starting from descendants to ancestors in the graph. Such an order is given by a reverse topological ordering. A topological ordering of a DAG, $G = (V, E)$, is an ordering \leq of its vertexes such that for every edge $v \rightarrow w$, $v \leq w$. Thus if we assume by this strategy that all vertexes 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 DFS of a DAG for printing the

vertexes in the reverse topological ordering. This ordering directly applies for the testing strategy.

C. General Case

However, in the general case, the TDG has cycles. The strategy thus has to consider them. This is done using a decomposition of the graph into its SCC and then breaking the SCC; which in turn implies the development of stubs.

1) *Strongly Connected Components:* A SCC of a graph is a subgraph such that for any pair of vertexes v, w , there is a path from v to w and vice versa. Two vertexes v, 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. The Tarjan algorithm [4] computes this DAG and prints SCC in the reverse topological ordering. The algorithm is not detailed here; some characteristics useful to our purpose are given.

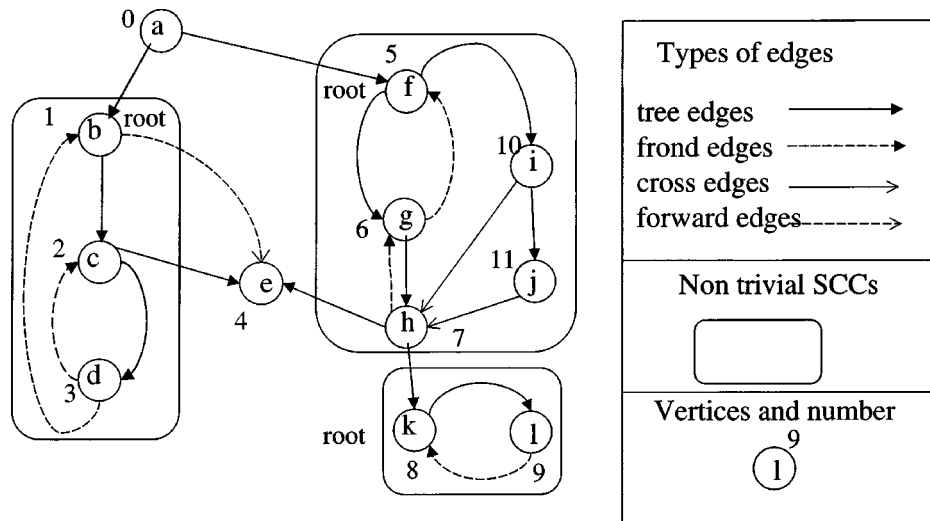


Fig. 6. Tarjan Algorithm and Partitioning of Edges

2) *The Tarjan Algorithm:* The Tarjan algorithm is based on a DFS with a supplementary stack that stores vertexes for which SCC is not completed. Vertexes are numbered according to their first visit by the DFS; let $v.num$ be this attribute. The DFS defines a partition of the edges into 4 classes (see Fig. 6):

- Tree edges lead from a vertex to an unvisited vertex. Tree arcs define a spanning forest of the graph.
- Forward edges are non tree-edges which go from a vertex to a descendant. These edges play no role in computing an SCC.
- Fronds go from a vertex to an ancestor in the spanning forest. A frond in a DFS is an edge which comes from a descendant to an ancestor.
- Cross edges are the remaining edges. They go from a vertex to a different subtree in the forest.

The root of an SCC is the first vertex of the SCC which is traversed during the DFS. The purpose of the algorithm is to detect roots of SCC. Thus it uses an attribute *lowlink* to vertex v which is the minimum $w.num$ of vertexes w which are accessible from v by tree edges and at most 1 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.

Because the reduced DAG of SCC is a DAG, we can use the same scheme as the strategy for the acyclic case. The inverse of the topological ordering given by the Tarjan Algorithm gives an ordering in which components in SCC can be tested for integration.

3) *The Bourdoncle Algorithm:* The Bourdoncle algorithm [5] is an adaptation of the Tarjan algorithm which was originally designed for static analysis and which can help in finding these stubs and to minimize their number. The main idea is to apply the Tarjan algorithm recursively to each non-trivial SCC, starting from its root, after having removed all fronds that enter the root. “Removing these edges” breaks some cycles in an SCC but not necessarily all of them. This is why the Tarjan algorithm must be recursively applied until all cycles are broken. The Tarjan algorithm has complexity quadratic in the size of

the graph. In the example of Fig. 7 the first call to Tarjan decomposes the graph into 5 SCC:

$$\{\underline{a}\}, \{\underline{b}, c, d\}, \{\underline{e}\}, \{\underline{f}, g, h, i, j\}, \{\underline{k}, l\},$$

with roots underlined. Trivial SCC

$$\{\underline{a}\}, \{\underline{e}\},$$

have no cycle, thus nothing has to be done. For non-trivial SCC, fronds entering the root are removed. For example $d \rightarrow b$ is removed from $\{\underline{b}, c, d\}$. The recursive call to the Tarjan algorithm then gives 2 SCC:

$$\{\underline{b}\}, \{\underline{c}, d\}.$$

Again frond $d \rightarrow c$ is removed from the only non-trivial component and the last call gives 2 trivial SCC

$$\{\underline{c}\}, \{\underline{d}\}.$$

The same decomposition is applied to

$$\{\underline{f}, g, h, i, j\}, \{\underline{k}, l\}.$$

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 Fig. 7. Roots of SCC in recursive calls are identified by squares; the SCC are surrounded. The partial ordering defined by this decomposition is given by the right part of Fig. 7.

Now this Tarjan algorithm is interpreted in terms of testing strategy. Because the decomposition is a partial ordering, there are several possibilities. For example, we can start by testing component l . Because l is the source of a frond to a root, it needs the target of the frond, k , as a stub. When l has been tested using stub(k), then 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 proceed by the left or right branch in any order. For the test of d , stub(b) and stub(c) are needed because $d \rightarrow b$ and $d \rightarrow c$ 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). Then j is tested using h ,

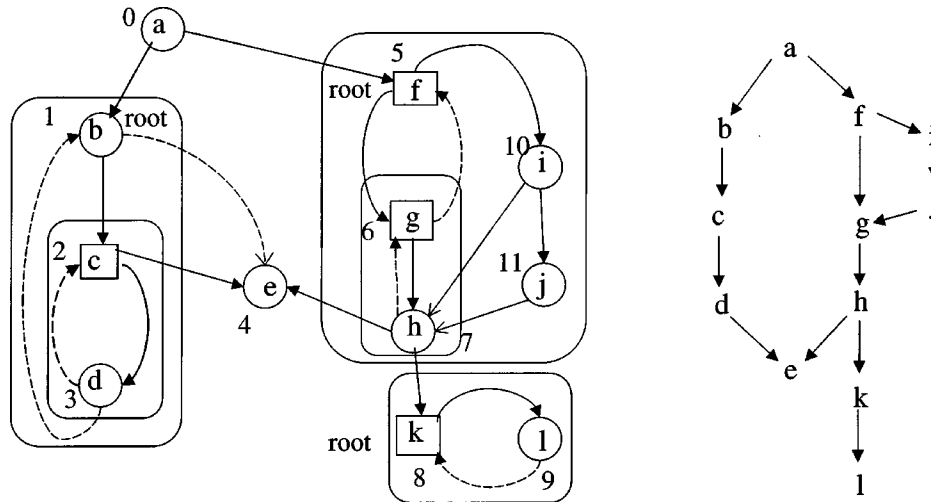


Fig. 7. Ordering Given by the Bourdoncle Algorithm

and then i is tested using j and h , and finally f is tested using g and i . The root node a can then be tested using b and f .

D. 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 Bourdoncle algorithm was different from ours, the strategy is not ideal for minimizing stubs. We propose a slightly different strategy where a stub should break as many cycles as possible. Finding the optimal solution for breaking cycles in an SCC is exponential while taking the root is of constant complexity. So, to keep the same complexity as in the Bourdoncle 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 outgoing fronds.

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

The application of the algorithm to the example of Fig. 6 is detailed in Fig. 8. The first call identifies the 3 non-trivial SCC. In SCC $\{b, c, d\}$ the selected vertex is d . In the recursive call, $c \rightarrow d$ is deleted and the remaining subgraph is acyclic. In $\{f, g, h, i, j\}$, the vertex g is selected, edges $f \rightarrow g$ and $h \rightarrow g$ are deleted; 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 $\text{stub}(k)$,
- k is tested,
- e is tested.

For the SCC $\{b, c, d\}$,

- c is tested using $\text{stub}(d)$ and e ,
- b is tested using c and e ,

- d is tested using c .

For $\{f, g, h, i, j\}$,

- h is tested first using $\text{stub}(g)$, e , and k ,
- j is tested using h ,
- i is tested using h ,
- f is tested using $\text{stub}(g)$ and i ,
- g is tested using h and f .

Now,

- a is tested using b and f .

Compared with the Bourdoncle algorithm, this algorithm gives better results on this example. In terms of cost of the algorithm, there are only 2 levels of recursive calls, but 3 for the Bourdoncle algorithm. This is because the our algorithm selects vertexes which break more cycles. In terms of effort for stubs, it is also better. There are only 3 realistic stubs (k, d, g) or 4 specific stubs corresponding to edges $l \rightarrow k, c \rightarrow d, f \rightarrow g, h \rightarrow g$. The Bourdoncle algorithm gives 5 realistic stubs (k, b, c, f, g) or 5 specific stubs corresponding to edges $l \rightarrow k, d \rightarrow c, d \rightarrow b, h \rightarrow g, g \rightarrow f$.

E. Regression Strategy

1) *General*: For regression, the previous optimized strategy, SIT, 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: coverage of each dependency path or from only direct dependencies. Moreover, when planning regression testing, contractual/implementation aspects should be used in deciding whether only CD/CCD are re-tested or ID also. If the application is not safety-critical, a weak regression strategy can be applied (contractual aspects and direct dependencies). This leads to the following coverage criteria (in the sense of Weyuker criteria [6]) which can guide the application of the algorithm for reducing the testing effort.

2) *Regression Test Adequacy Criteria*: For testing a component C in a system, two basic criteria can be used.

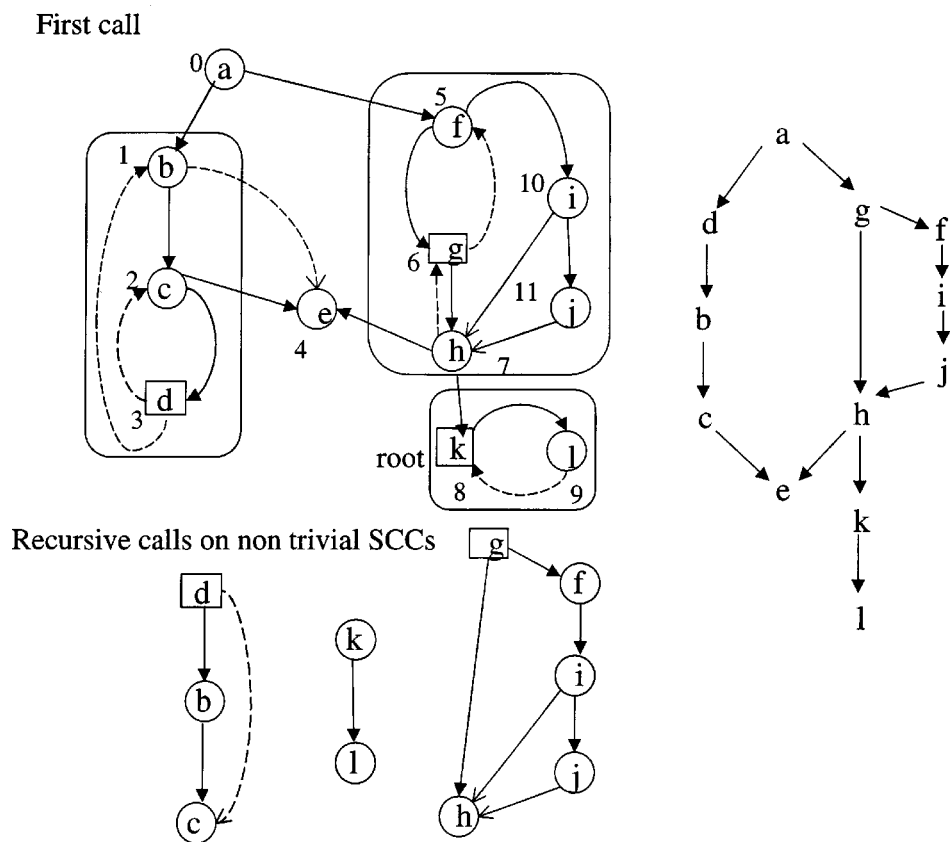


Fig. 8. Optimized Algorithm and Corresponding Ordering

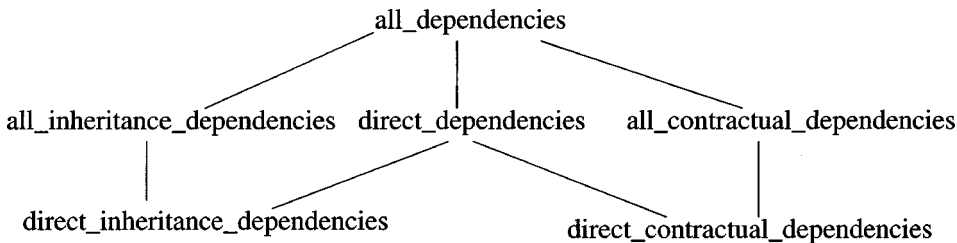


Fig. 9. Partial Ordering Between Test Adequacy Criteria

- 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 can be applied to each type of dependency (contractual inheritance, contractual client, and implementation dependencies). A simple partial ordering relationship exists between adequacy criteria as presented for contractual dependencies in Fig. 9 (the stronger criterion is on the top of the figure).

IV. CASE STUDY AND RESULTS

A case study from the telecommunication domain makes this paper more concrete. 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 is in [7]. Fig. 10 is its UML class diagram. Each class has a number which corresponds to a node in the TDG (Fig. 11). It is composed only by class-to-class test contractual TD.

Our purpose is to produce the test plan for integration testing and to compare the stubs gain with various strategies, including the presented optimized strategy, SIT. Because minimizing test stubs is an NP-complete problem, SIT is an efficient heuristic which produces a result close to the true minimum. The stub counting is performed both with specific and realistic stubs. It provides a first estimate of the integration testing effort.

The strategies are listed here; they have been applied directly and after a first decomposition of the TDG into connected parts using the Tarjan algorithm.

- RC (random component selection): The components are tested in a uniform random order.
- MC (most used component selection): The components are tested from the most used (maximum of predecessors)

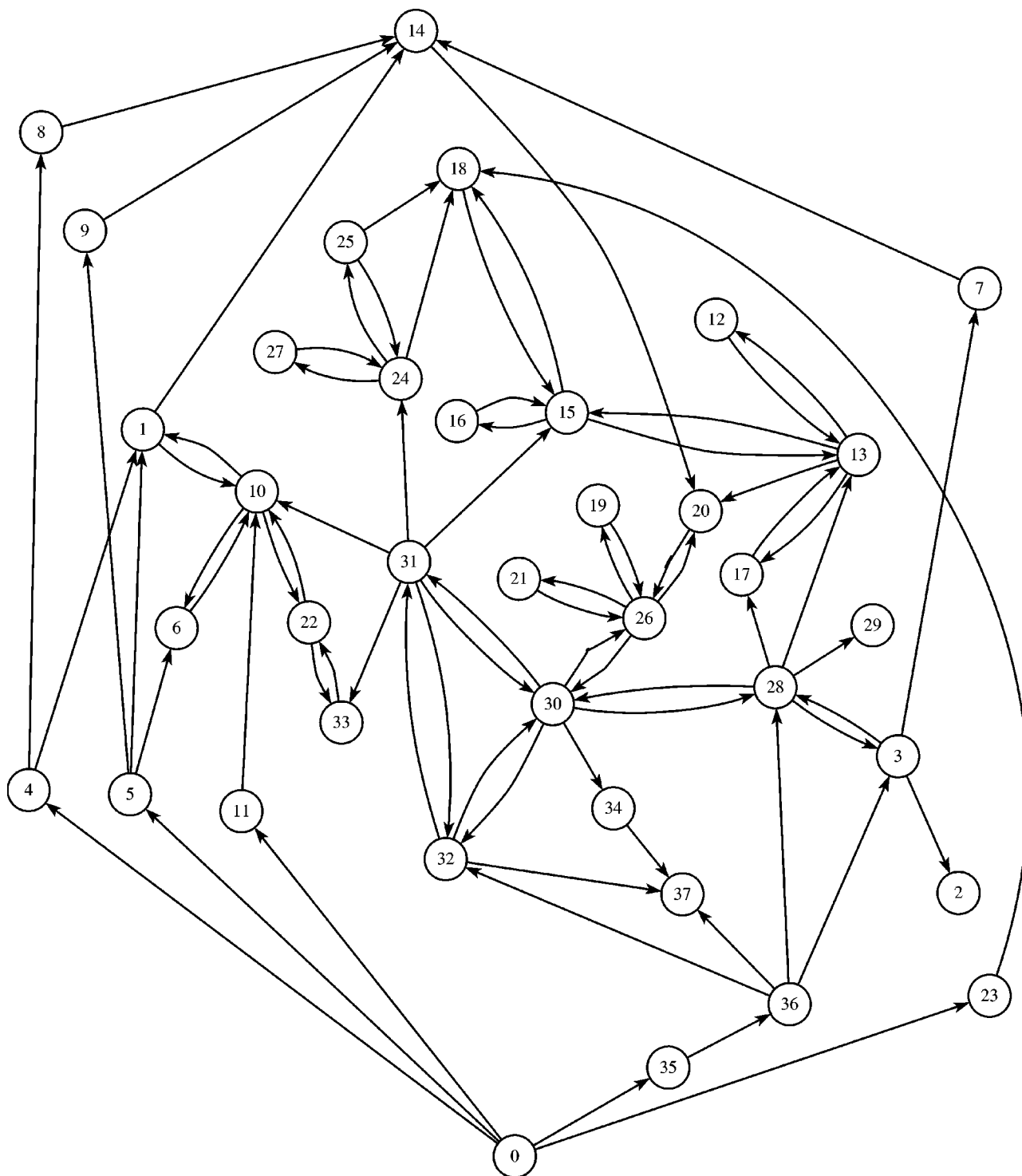


Fig. 11. TDG of the SMDS Case-Study

that one tester resource had to organize the integration. This section generalizes the approach to n -testers resources.

A tester is defined as: a testing team of a given, fixed size. It represents a given unit testing effort that can be allocated to a testing task.

The following assumption helps to present the test repartition strategy:

- a tester needs “1 time unit or step” to integrate a component to the system and test it.

The following strategy can be easily adapted by giving a weight to each node of the graph corresponding to the component estimated complexity.

The problem is thus defined as:

- Given n -testers testing resources, what is the best way to allocate “component integration testing tasks” to “minimizing the integration duration” *viz.*, number of steps needed to achieve system integration?

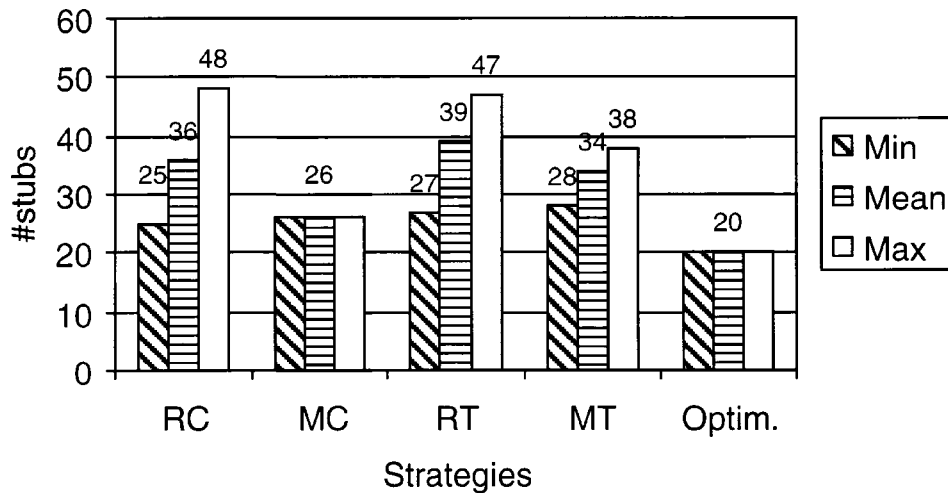


Fig. 12. Specific Stubs Counting

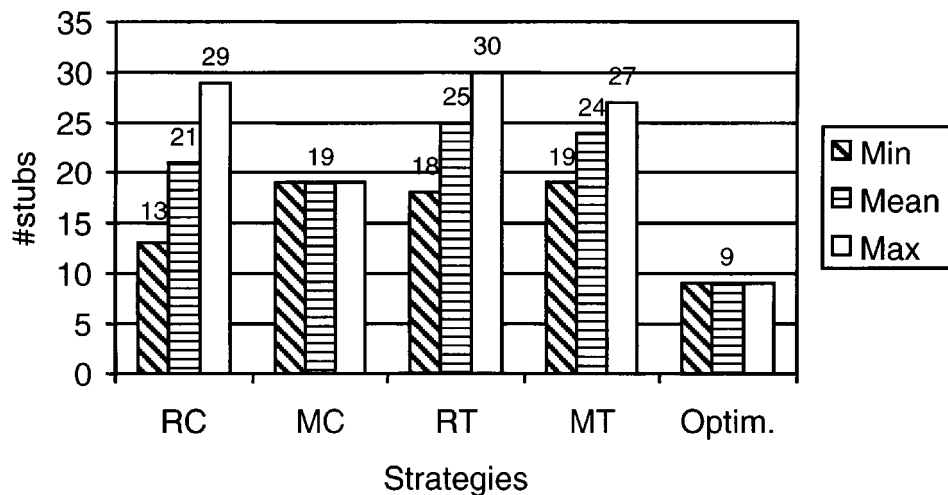


Fig. 13. Realistic Stubs Counting

Notation

- #steps_min: minimum number of steps required for integration,
- #nodes: number of nodes (or components) of the partial order graph,
- #testers: number of available testing resources that can be allocated to testing tasks,
- path_length: maximum length (number of nodes included in a path) of paths in the partial order graph.
- A : #nodes/#testers + 1
- B : path_length

Then: #steps_min = max(A , B)

This property can be easily demonstrated since B is a bound on the minimum number of steps: the steps on a given path cannot be parallelized between testers. A is the global number of tasks that can be optimistically divided between testers.

The repartition strategy is:

- part 1
Assign a value to each node equal to the maximum length of partial paths beginning on it.
- part 2:

```

From n_steps = 0
until all nodes have been allocated
repeat
  n_steps = n_steps + 1
  allocate nodes which have the maximum
  value to each tester;
stop when all nodes have been allocated
end

```

The complexity of “part 1” and “part 2” is linear with the number of nodes. The global heuristic is thus linear.

Table I presents the resulting repartition of each integration step between 4 testers for the SMDS case study. This result reaches the theoretical #steps-min value expressed in the property.

- number of nodes is 37,
- number of testers is 4
- length of longest path is 8.

Because a simple graph model is available, the objective of a test integration plan has been easily continued by providing an

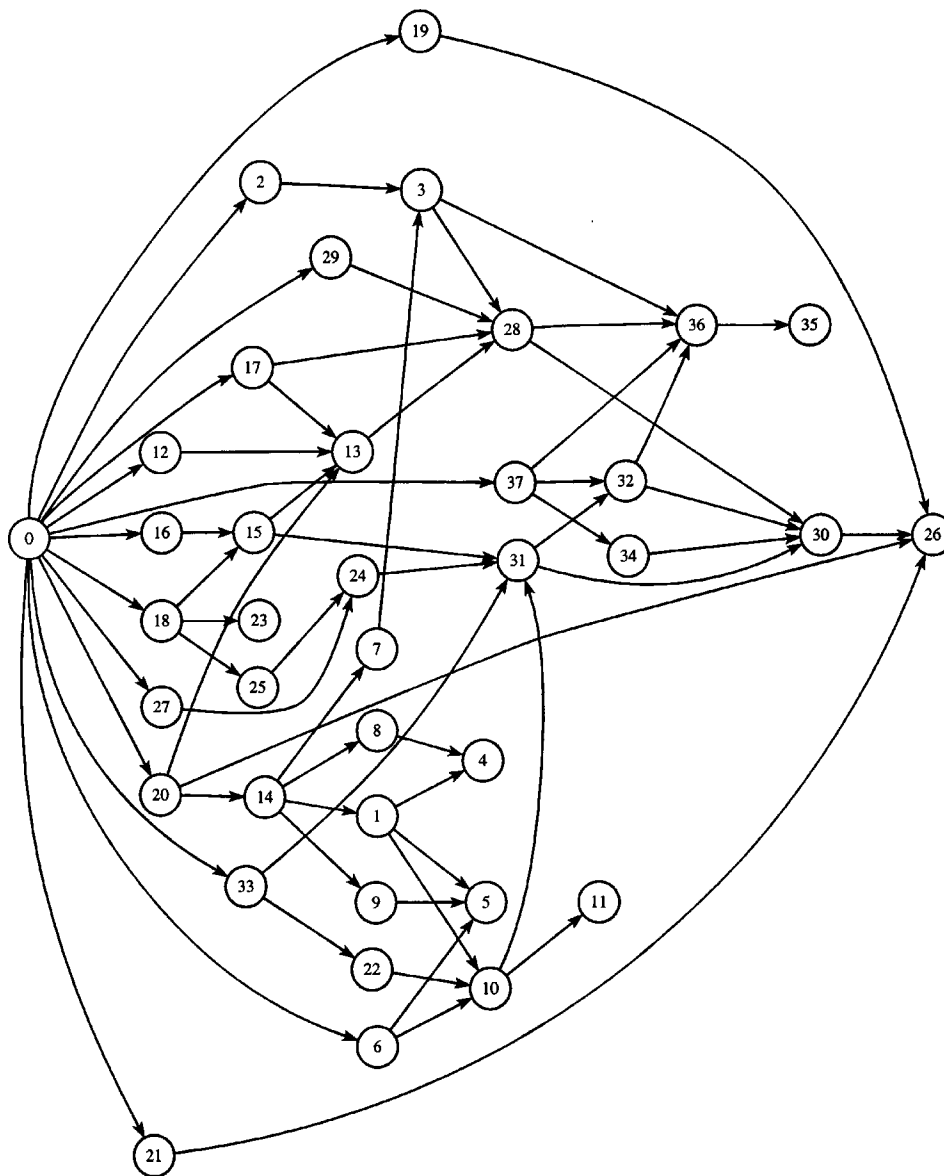


Fig. 14. Partial Order Graph

TABLE I
TEST REPARTITION FOR THE SMDS CASE STUDY (4 TESTERS)

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

efficient way of “allocating testing resources” and “minimizing integration duration.” This also provides a first analysis of the way to minimize testing costs by improving test planning and parallelizing testing tasks.

VI. RELATED WORKS

Very few of the numerous first-generation books on analysis, design, and implementation of OO software explicitly address validation and verification issues. Despite this initial lack of interest, testing of OO systems is receiving much more attention; see [8] for a detailed state of the art.

Concerning OO testing techniques, most of the work focuses on the dynamic aspects of OO systems: a system is viewed as a set of cooperating agents, modeling objects, and is modeled with FSM, or equivalent object-state modeling [9]–[11]. Such works deal with limitations concerning computation cost of mapping object behavior into the underlying model. One solution decomposes the program into hierarchical and functionally-coherent parts. In such approaches, this decomposition provides a framework for unit, integration, and system test definition. In [12], 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 [11], [13] approaches. In particular, [11] 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 regression, integration, and maintenance purposes. This paper differs in the way of organizing a 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 other strategies;
- testing resources repartition: to our knowledge, no existing works are available applied to early-design stage;
- test methodology: our 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 [15]. The test plan evolves during all design refinement steps (including code production).

REFERENCES

- [1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Guide*: Addison-Wesley, 1998.
- [2] B. Beizer, *Software Testing Techniques*: Van Nostrand Reinhold, 1990.
- [3] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*: Addison-Wesley, 1998.
- [4] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Computing*, vol. 1, pp. 146–160, June 1972.
- [5] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *Proc. Int'l Conf. Formal Methods in Programming and Their Applications*, 1993, Lecture Notes in Computer Science 735, pp. 128–141.
- [6] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Engineering*, vol. 11, pp. 367–375, 1985.
- [7] J.-M. Jézéquel, *Object Oriented Software Engineering with Eiffel*: Addison-Wesley, 1996.
- [8] R. V. Binder, "Testing object-oriented software: A survey," *J. Software Testing, Verification, Reliability*, vol. 6, pp. 125–252, 1996.
- [9] —, "Design for testability with object-oriented systems," *Communications ACM*, vol. 37, pp. 87–101, Sept. 1994.
- [10] P. C. Jorgensen and C. Erickson, "Object-oriented integration testing," *Communications ACM*, vol. 37, pp. 30–38, Sept. 1994.
- [11] D. C. Kung, J. Gao, and C. Chen, "On regression testing of object-oriented programs," *J. Systems and Software*, vol. 32, Jan. 1996.
- [12] J. D. McGregor and T. Korson, "Integrating object-oriented testing and development processes," *Communications ACM*, vol. 37, pp. 59–77, Sept. 1994.
- [13] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structures," in *Proc. 14th Int'l. Conf. Software Engineering*, May 1992, pp. 68–80.
- [14] K.-C. Tai and F. J. Daniels, "Interclass test order for object-oriented software," *J. Object-Oriented Programming*, pp. 18–35, July–Aug. 1999.
- [15] Y. Le Traon, D. Deveaux, and J.-M. Jézéquel, "Self-testable components: From pragmatic tests to a design-for-testability methodology," *Proc. TOOLS—Europe'99*, pp. 96–107, June 1999.

Yves Le Traon received the Engineering degree in Computer Science from the Ecole Nationale Supérieure d'Informatique et Mathématiques de Grenoble (ENSIMAG), and a Ph.D. (1997) in Computer Science from the Institut National Polytechnique de Grenoble, France. He is an Assistant Professor at the University of Rennes I. His research interests concern software quality measurements, testability and object-oriented testing.

Thierry Jérón has his Ph.D. (1991) in Computer Science from the University of Rennes. His work was on protocol verification. He spent 1 year in the Alcatel Research Laboratory (1992–1993). Since 1993, he has been a full INRIA researcher at Irisa in Rennes. His main research interests are verification by model-checking and automatic test-generation for reactive systems. He is the main contributor to the automatic test generation tool, TGV.

Jean-Marc Jézéquel received an Engineering degree (1986) in Telecommunications from the ENSTB, and a Ph.D. (1989) in Computer Science from the University of Rennes, France. He then worked for the TRANSPAC company on an Intelligent Network project. He is now a research manager in the Irisa Lab for the Centre National de la Recherche Scientifique. His research interests include software-engineering and object-oriented technologies for telecommunications and distributed computers. He is the author of the books *Object-Oriented Software Engineering with Eiffel*, 1996, and *Design Patterns with Contracts*, 1999, and of more than 40 publications in international journals and conferences.

Pierre Morel received the Engineering degree (1996) in Computer Science from the Institut de Formation Supérieure en Informatique et Communication (IFSIC). He is a Ph.D. student in Computer Science in the domain of Conformance Testing. He is co-author of the prototype tool TGV which automatically generates adaptive test-cases using on-the-fly model-checking technology for reactive systems.