Testing Levels for Object-Oriented Software

Y. Labiche   P. Thévenod-Fosse   H. Waeselynck
LAAS-CNRS/LIS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4
FRANCE
{ylabiche, thevenod, waeselyn}@laas.fr

M.-H. Durand
Aerospatiale Matra Airbus/LIS
Département Systèmes (BTE/SY/MS)
316 route de Bayonne
31060 Toulouse Cedex 3
FRANCE
m-helene.durand@airbus.aeromatra.com

ABSTRACT
One of the characteristics of object-oriented software is the complex dependency that may exist between classes due to inheritance, association and aggregation relationships. Hence, where to start testing and how to define an integration strategy are issues that require further investigation. This paper presents an approach to define a test order by exploiting a model produced during design stages (e.g., using OMT, UML), namely the class diagram. Our goal is to minimize the number of stubs to be constructed in order to decrease the cost of testing. This is done by testing a class after the classes it depends on. The novelty of the test order lies in the fact that it takes account of: (i) dynamic (polymorphism) dependencies; (ii) abstract classes that cannot be instantiated, making some testing levels infeasible. The test order is represented by a graph showing which testing levels must be done in sequence and which ones may be done independently. It also provides information about the classes involved in each level and how they are involved (e.g., instantiation or not). The approach is implemented in a tool called TOONS (Testing level generator for Object-Oriented Software). It is applied to an industrial case study from the avionics domain.

Keywords
Object-oriented software, test order, integration strategy, polymorphism, abstract classes, tool.

1 INTRODUCTION
It is now well recognized that testing approaches must be revisited to account for the specific characteristics of object-oriented (OO) technology (see e.g., [2], [9]). This paper addresses one of the problems raised by OO technology, namely the determination of testing levels.

Testing levels correspond to subsets of the target program to be tested in a certain order, thus defining an integration process from unit to system testing.

The "traditional" unit and integration levels defined for procedural programs do not fit well in the case of OO programs. Unit testing of subprograms cannot easily be mapped onto testing of individual object's operations: taken in isolation, the body of one operation typically consists of a few lines of code; its behavior is meaningless unless analyzed in relation to other operations and their joint effect on a shared state. Hence, any significant unit to be tested typically cannot be smaller than the instantiation of one class. Moreover, there is the general problem of the complex dependency between classes. As pointed out in [8], the many relationships that exist in an OO program imply that one class inevitably depends on other class(es). It is difficult to determine where to start testing, and there is no obvious order for an integration strategy. The determination of testing levels may depend on:

1. The possibility of associating a functional description with a (set of) class(es). The instantiation of the class(es) must form a subsystem that is sufficient to generate meaningful behavior patterns. For example, in [6], the integration process is explicitly based on behavioral considerations: gray box analysis is used to identify the most significant paths and interactions, which yields the testing levels. In [10], the definition of software increments to be gradually integrated is based on a structured model of use cases.
2. The number of stubs to be constructed. Every server class not included in the target piece of software has to be simulated, which increases the cost of testing. The integration strategy proposed by Kung et al. [7] defines a class test order according to the analysis of class dependencies, so that the number of stubs is minimized.

Like the authors of [7], we believe that issue 2 is of utmost importance when proposing an integration strategy. First, it is not always feasible to construct a stub that is simpler than the piece of code it simulates. Second, there is no hope for
stub generation to be fully automated, because it requires understanding of the semantics of the simulated functions. Third, the bug potential for some stubs may be higher than that of the real function [1]. As a result, minimizing the number of stubs to be constructed should result in drastic savings.

Adopting this concern as a guideline, our aim is to identify the testing levels early in the development process. This should give a first idea of the complexity of the integration process, when design choices may still be reconsidered for testing purposes. Accordingly, the paper presents an approach exploiting a model produced during design stages (e.g., using OMT, UML, ...). It focuses on the definition and partial ordering of testing levels based on the class diagram. Note that the problem of test data selection at each level is out of the scope of the paper.

Since the number of stubs is a concern shared with [7], our work started with the analysis of their approach (see Section 2). Their idea is that the class(es) under test at a given level should depend only on classes previously tested. Hence, no stub is required since the previously tested class(es) can be included in subsequent testing levels. Originally defined for regression testing purposes, their class dependency analysis is based on a model obtained by reverse engineering. But since the model is similar to an UML (or OMT, ...) class diagram, the approach can also be applied during design stages. Indeed, our work can be seen as an improvement on the topological order proposed by Kung et al. The main difference is that we take into consideration polymorphism and potential dynamic binding features, while Kung et al.'s approach is purely static. We also propose modifications of the integration process to account for abstract classes. Moreover, our representation of testing levels displays synthetic information about the role of each class within a given subsystem. The whole method is presented in Section 3, as well as the tool that implements it. Then, using our tool, the method is exemplified by a case study (Section 4) coming from the avionics domain. Section 5 further comments on related work. Section 6 concludes on our on-going work.

2 KUNG ET AL.'S TEST ORDER

The test order finding algorithm defined by Kung et al. [7, 8] is based on a class model called the Object Relation Diagram (ORD). The ORD for a program P is an edge-labeled digraph where the nodes represent the classes in P, and the edges represent the inheritance, aggregation and association relations. For any two classes C1 and C2:

- An edge labeled I from C1 to C2 indicates that C1 is a child class of C2;
- An edge labeled As from C1 to C2 indicates that C1 is an aggregated class of C2 (C1 contains one or more objects of C2);
- An edge labeled As from C1 to C2 indicates that C1 associates with C2.

Thus, the ORD captures the static – i.e., at compile time – dependencies between the classes. For example, Figure 1a shows the ORD for a hypothetical program with 8 classes \{A, ..., H\}: A is both a parent class of E, and a component class of B; class B associates with C; and so on.

Kung et al. present a method for constructing an ORD by reverse engineering from C++ programs. Then they define the notion of class firewall to identify the effect of a class change at the class level. Finally, a test order finding algorithm is proposed to provide the tester with a road map for conducting retests of classes in the firewall. Based on these results, an OO testing environment has been developed using the reverse engineering approach [8].

Sections 2.1 and 2.2 briefly recall Kung et al.'s approach. Then, we comment on it to introduce the motivation for our work.

2.1 Class Firewall

The class firewall, denoted CFW(X) for a class X, is the set of classes that could be affected by changes to class X and thus, that should be retested when class X is changed. Assuming that the relationships between classes are not affected by the changed class (the ORD for the program is not modified), Kung et al. argue that, for adequate testing, CFW(X) has to include: the child classes of X, the aggregated classes of X, and the classes associated with X. Hence, the algorithm for computing a class firewall is transitively derived using the dependence binary relation R deduced from the directed edges of an ORD:

![Diagram](image)

(a) ORD

(b) Class firewalls

(c) Test order

Fig. 1. Example 1
\[ R = \{ \langle C_i, C_j \rangle \mid \text{there is a directed labeled edge from } C_i \text{ to } C_j \text{ in the ORD} \} \]

\( R \) relates each class to the classes that depends on it through a direct inheritance, aggregation, or association relationship. Then, \( CFW(X) = \{ C_k \mid \langle X, C_k \rangle \in R^* \} \), where \( R^* \) is the irreflexive transitive closure of \( R \). \( CFW(X) \) contains all the classes \( C_k \) such that there is a directed path from \( C_k \) to \( X \) in the ORD (see e.g., Fig. 1b).

The notion of a class firewall is extended to a set \( S = \{ X_1, \ldots, X_q \} \) of changed classes: the class firewall for \( S \) is defined as \( CFW(S) = CFW(X_1) \cup \cdots \cup CFW(X_q) \).

### 2.2 Test Order for Class Firewalls

The test order problem for class firewalls is stated by Kung et al. as finding a desirable order for testing the classes affected by code changes to a set of classes. Here, a desirable test order is one that requires minimum effort to construct the test stubs. Assuming that the number of stubs is a good (though not perfect) estimator for effort, this number has to be minimized. Hence, the main idea is to test the independent classes first and then test the dependent classes based on their relationships. For example, testing the component classes before the classes that contain them allows the tester to use the actual component classes instead of constructing test stubs. Similarly, a child class should be tested after its parent class(es); then, the test information for the parent class can be reused in testing the child classes, as suggested in [4] for example.

Based on this principle, a test order finding algorithm is presented which applies to acyclic as well as cyclic ORDs. In the case of an acyclic ORD, the algorithm produces a topological sorting. This test order ensures that \( X \) is tested before all the classes of \( CFW(X) \). Hence, stubs are not required. For example, let us consider the acyclic ORD in Figure 1a and assume that code changes are performed in both classes A and D: then classes A, D and the other six classes have to be retested, since from Figure 1b, \( CFW(\{A, D\}) = \{B, C, E, F, G, H\} \). Figure 1c shows the test order that involves four successive testing levels: when several classes have the same level number, either of them can be tested before the other. For example, classes A and D have to be tested first, in any order; E, F and H have to be tested after the other five classes in any order.

In the case of a cyclic ORD, topological sorting cannot be applied. The Kung et al.'s algorithm is based on two concepts: first, the notion of cluster, which is a maximal set of nodes that are mutually reachable through the relation \( R \); the second notion is cycle breaking, that is, identifying and removing an edge(s) from a cluster until the graph becomes acyclic. The algorithm consists of: (i) transforming the cyclic ORD into an acyclic digraph ORD' where the nodes represent clusters; topological sorting is applied to produce a test order for the clusters; (ii) for each cluster, breaking the cycle and using topological sorting to produce a test order for the classes in the cluster. To break a cycle, the authors suggest to (temporarily) remove one association edge because the association relation represents the weakest coupling between two related classes. Such a solution is always possible since, as demonstrated by the authors, every directed cycle of an ORD contains at least one association edge. We do not further introduce the method since the focus of our work is to propose improvements on the topological sorting as a basis for defining a test order in cases of acyclic (sub)digraph, that is, after cycle breaking.

### 2.3 Comments and Motivation

Although initially defined for regression testing purposes, the test order finding algorithm may be applied to a new software system, thus determining the testing levels for it. Moreover, the information contained in the ORD is similar to the one provided by the class diagram produced during an OO development process (e.g., UML): instead of constructing the ORD by reverse engineering, one can derive it from design documents to define a test order early in the development process. But, some limitations may be pointed out that require further improvement. In particular:

1. **Dynamic** — i.e., *at execution time* — relationships between classes are not taken into account. For example (Fig. 1a), class F associates with G which is a parent class of H; then, due to polymorphism, F may dynamically associate with H and thus, should be tested after H; F and H should not be assigned the same testing level as stated in Figure 1c.

2. The test order provides information on the classes to be tested at each level. But, it does not give explicit information on the other classes that have to be involved in the test experiments, that is, the classes on which the class under test is dependent. Such information should allow early analysis of the design process, while design choices are still open. It may also be used to define the test criteria at each level.

3. Some test levels become (partly) infeasible in cases of abstract classes so that the test order should be revised. For example (Fig. 1), let us assume that A is an abstract class; since A cannot be instantiated, testing it at level 1 is unworkable; but the important thing is that testing B at level 3 now requires child class E to be instantiated (instead of A); yet, E is planned to be tested after B since it associates with B.

The three points listed above are the main motivation for the method proposed in Section 3. We will return to the comparison with Kung et al.'s test order in Section 5.

### 3 METHOD FOR DEFINING AND ORDERING THE TESTING LEVELS

The method involves five steps. First, both static and dynamic dependencies between the classes are analyzed (Section 3.1). The second step consists of defining the testing levels from the results of the analysis (Section 3.2); the definition includes information about the class(es) under test, the set of classes involved in the level, and the kind of dependencies taken into account. Third, a partial ordering relation is defined to produce a test order: the
result is a graph which shows the precedence relations between the testing levels (Section 3.3). Furthermore, the definition of test criteria to be associated with each testing level is prepared by indicating the role of each class involved in the level (Section 3.4). Finally, infeasible testing levels due to abstract classes are removed from the graph (Section 3.5). The five steps of the method are implemented in a software tool (Section 3.6).

3.1 Class Dependencies Analysis

Two sets of classes and a Boolean function are associated with each class X:

- \( D_1(X) \) is the set of classes on which X depends statically, i.e. at compile time;
- \( D_2(X) \) is the set of classes on which X depends either statically or dynamically (at execution time) or both;
- the Boolean function denoted \( B_d(X) \) indicates whether or not X may dynamically depend on at least one class of \( D_2(X) \), due to polymorphism.

The method for computing \( D_1(X) \), \( D_2(X) \), and \( B_d(X) \) starts with a description of the static dependencies as shown in an ORD. As said in Section 2.2, we assume that cycles in the ORD have already been broken. Yet, we will see that cycles may exist while considering dynamic dependencies in addition to static ones.

Definition of the Sets \( D_1(X) \)

A class X statically depends on a class \( C_k \) if and only if there is a directed path from X to \( C_k \) in the ORD. Let \( R_s \) be the binary relation associated with the directed edges of an ORD: \( R_s = \{ <C_1, C_2> | \text{there is an edge from } C_1 \text{ to } C_2 \text{ in the ORD} \} \). Then, \( D_1(X) = \{ C_k | <X, C_k> \in R_s \} \). For example (see Fig. 1a): \( D_1(F) = \{ C, D, G \} \). Note that \( R_s \) is the inverse relation of the binary relation \( R \) defined by Kung et al. to compute the class firewall \( CFW(X) \), because \( CFW(X) \) is the set of classes which depend on X.

Definition of the Sets \( D_2(X) \)

Dynamic dependencies due to polymorphism can be derived from static dependencies. For example (see Fig. 1a): Class B associates with class C which is a parent class of classes F, G and H; then, at execution time, class B may associate with classes F, G and H.

More generally, if \( C_j \) is a server class of class X – i.e., X associates with \( C_j \) or is an aggregated class of \( C_j \) – then, at execution time, X may depend on all the (direct or by transitivity) child classes of \( C_j \). Hence, the binary relation \( R_d \) defined below represents the dynamic dependencies:

\[ R_d = \{ <C_1, C_2> \mid \exists C_k \text{ such that } C_k \text{ is both a server class of } C_1 \text{ and a parent class of } C_2 \} \]

\( R_d \) is denoted by dotted directed edges which are added to the ORD. Figure 2a shows the completed \( ORD \), called \( C-ORD \), deduced from the ORD in Figure 1a.

A class X depends on a class \( C_k \) either statically or dynamically or both if and only if there is a directed path from X to \( C_k \) in the \( C-ORD \), i.e.:

\[ D_2(X) = \{ C_k | <X, C_k> \in (R_s \cup R_d) \} \]

Figure 2b tabulates the sets \( D_1(X) \) and \( D_2(X) \) calculated for Example 1. Note that, although we consider only acyclic ORDs, dynamic dependencies can introduce cycle(s) in the \( C-ORD \), that is, we may have X \( \in D_2(X) \). In that case, all the classes in the cycle have the same set \( D_2(X) \). For example, the dotted edge from B to E introduces a cycle between B and E, which leads to \( D_2(B) = D_2(E) \) and both classes B and E belong to this set.

Boolean Function \( B_d(X) \)

By construction, \( D_1(X) \subseteq D_2(X) \) for every class X. If \( D_1(X) \) is a strict subset of \( D_2(X) \), it implies that X dynamically depends on some class(es) of \( D_2(X) \) while the reverse is not true: X may dynamically depend on some class(es) of \( D_2(X) \) even if \( D_1(X) = D_2(X) \). For example, let us consider class \( \gamma \) in Figure 3: \( \gamma \) statically depends on both \( \beta \) and \( \alpha \); since \( \beta \) may dynamically associate with itself, polymorphism involves additional relationships between \( \gamma \) and \( \beta \). Yet, \( D_1(\gamma) = D_2(\gamma) \).

![Diagram](image)

(a) C-ORD

(b) \( D_1(X), D_2(X) \) and \( B_d(X) \)

---

Fig. 2. Example 1: Analysis of static and dynamic dependencies

<table>
<thead>
<tr>
<th>Class X</th>
<th>( D_1(X) )</th>
<th>( D_2(X) )</th>
<th>( B_d(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>( {A,C,D} )</td>
<td>( {A,B,C,D,E,F,G,H} )</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>( {D} )</td>
<td>( {D} )</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>( {A,B,C,D} )</td>
<td>( {A,B,C,D,E,F,G,H} )</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>( {C,D,G} )</td>
<td>( {C,D,G,H} )</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>( {C,D} )</td>
<td>( {C,D} )</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>( {C,D,G} )</td>
<td>( {C,D,G} )</td>
<td>0</td>
</tr>
</tbody>
</table>
More generally, class $X$ dynamically depends on other class(es) if and only if at least one class $C_k$ of $D_2(X) \cup \{X\}$ has a dotted outgoing edge in the C-ORD. In that case, the Boolean function $B_d(X)$ which captures the existence of dynamic dependencies has value 1; otherwise, $B_d(X) = 0$. Note that, if $D_1(X) \subset D_2(X)$ such a class $C_k$ does exist by the construction.

Figures 2b and 3b give the values $B_d(X)$ associated with the examples.

3.2 Testing Levels

To determine the testing levels, we adopt the following strategy: (i) for every class, a specific testing level is defined that accounts for static dependencies only, that is, dependencies at compile time; (ii) testing of additional dependencies that may occur at execution time, is the target of separate levels.

A testing level $T$ is then described by a triplet $(T.g\text{oal}, T.\text{need}, T.\text{type})$, where:

- $T.g\text{oal}$ is the set of classes under test;
- $T.\text{need}$ is the set of classes that have to be involved in the test experiment; it contains the classes under test ($T.g\text{oal}$) and all the classes on which they are dependent according to $T.\text{type}$;
- $T.\text{type}$ indicates the type of dependencies taken into account: either static (Sta) or dynamic (Dyn).

The levels are directly derived from $D_1(X), D_2(X)$ and $B_d(X)$ since: $D_1(X)$ gives the minimum set of classes required to test class $X$, that is, the set of classes needed at compile time (without taking polymorphic relationships into account); $D_2(X)$ gives the maximum set of classes required to test class $X$, when taking polymorphic relationships (if any) into account; and, $B_d(X)$ indicates whether or not a testing level with $T.\text{type} = \text{Dyn}$ is to be defined. Hence, the testing levels are built as follows.

A testing level $T = (\{X\}, \{X\} \cup D_1(X), \text{Sta})$ is associated with each class $X$. The first 8 rows of Table 1 show the testing levels obtained from the sets $D_1(X)$ in Figure 2b.

As regards dynamic dependencies, every class $X$ such that $B_d(X) = 1$ must be the focus of a testing level $(T.g\text{oal}, T.\text{need}, \text{Dyn})$. When $X$ is the class under test, $T.\text{need} = \{X\} \cup D_2(X)$ defines the classes to be involved in the test experiment. But, due to possible cycles ($X \in D_2(X)$), identical sets $T.\text{need}$ may be obtained for different classes (see Section 3.1). For example, classes $B$ and $E$ in Figure 2 lead to: $T.\text{need} = D_2(B) = D_2(E) \supseteq \{B, E\}$. In that case, a single testing level will be associated with the classes of a cycle, by grouping them in $T.g\text{oal}$. Hence, the following testing levels are defined:

- $(\{X\}, \{X\} \cup D_2(X), \text{Dyn})$, in case there is no cycle involving $X$;
- $(\{X_1, \ldots, X_j\}, D_2(X_j), \text{Dyn})$ otherwise, where $X_1, \ldots, X_j$ are all the classes involved in a same cycle.

Returning to the example in Figure 2b, we get two testing levels related to dynamic dependencies (see Table 1).

### Table 1: Testing levels for Example 1

<table>
<thead>
<tr>
<th>$T.g\text{oal}$</th>
<th>$T.\text{need}$</th>
<th>$T.\text{type}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${A}$</td>
<td>${A}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${B}$</td>
<td>${A,B,C,D}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${C}$</td>
<td>${C,D}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${D}$</td>
<td>${D}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${E}$</td>
<td>${A,B,C,D,E}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${F}$</td>
<td>${C,D,F,G}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${G}$</td>
<td>${C,D,G}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${H}$</td>
<td>${C,D,G,H}$</td>
<td>Sta</td>
</tr>
<tr>
<td>${B,E}$</td>
<td>${A,B,C,D,E,F,G,H}$</td>
<td>Dyn</td>
</tr>
<tr>
<td>${F}$</td>
<td>${C,D,F,G}$</td>
<td>Dyn</td>
</tr>
</tbody>
</table>

3.3 Partial Ordering of Testing Levels

The aim is to find a test order that takes account of (i) the dependencies between classes – a class is tested after the classes it depends on – as well as (ii) the kind of these dependencies – dynamic dependencies are considered after static ones. Issue (i) is intended to reduce the number of stubs to be constructed since the previously tested class(es) can be included in subsequent testing levels. Issue (ii) should allow the tester to solve static problems first, in order to facilitate testing of polymorphic relationships.

**Definition**

Let $M = (M.g\text{oal}, M.\text{need}, M.\text{type})$ and $N = (N.g\text{oal}, N.\text{need}, N.\text{type})$ be two testing levels. Then,

$$M \subseteq N \Leftrightarrow \begin{cases} M.g\text{oal} = N.g\text{oal} \text{ and } M.\text{need} \subseteq N.\text{need} \\
                            (M.\text{type} = \text{Sta} \text{ and } N.\text{type} = \text{Dyn}) \text{ and } \\
                            M.\text{need} \subseteq N.\text{need} \end{cases}$$

where $M \subseteq N$ means that level $M$ precedes level $N$.

**Properties**

We have proved that $\subseteq$ is a partial order and that it satisfies the following four properties:

1. If class $X$ statically depends on class $Y$, then testing level $(\{Y\}, \{Y\} \cup D(X), \text{Sta})$ precedes testing level $(\{X\}, \{X\} \cup D(X), \text{Sta})$. 
(2) If class X necessitates two testing levels \(B_2(X) = 1\), then testing level \((X), \{X\} \cup D_2(X), \text{Sta}\) precedes testing level \((\text{goal}, \{X\} \cup D_2(X), \text{Dyn})\) with \(X \in \text{goal}\).

(3) Let X be a class necessitating a testing level Tx with type Dyn, and Y ∈ Tx.need with Y ≠ X. Then, \((\{Y\}, \{Y\} \cup D_2(Y), \text{Sta})\) precedes Tx.

(4) Let X and Y be two classes necessitating distinct testing levels with type Dyn, say Tx and Ty. If X dynamically depends on Y, then Ty precedes Tx.

Note that the properties meet both original requirements of the test order (see issues (i) and (ii) above).

**Graphical Representation**

The test order is represented by a graph \(G = (V, E)\) where the vertices \(V\) are the testing levels and the edges \(E\) indicate immediate precedences. The graph shows which testing levels must be treated in sequence and which ones may be treated independently. For example, \((\{1\}, \{1\}, \text{Sta})\) in Figure 4a precedes \((\{C\}, \{C\}, \text{Sta})\) and may be treated independently from \((\{1\}, \{2\}, \text{Sta})\).

Note that owing to property (1), all the classes are introduced one by one throughout the subgraph composed of the testing levels of type Sta. For example, let us consider the level \((\{B\}, \{A,B,C,D\}, \text{Sta})\): the goal B is the only class which was not involved in at least one of the two immediate preceding levels. When it comes to testing dynamic dependencies, no new class is introduced (owing to properties (2), (3), and (4)).

### 3.4 Adding Information to Testing Levels

The triplet representation of testing level T explicitly shows which classes are the target of testing (T.goal) and whether or not polymorphism is to be taken into account (T.type). However, this representation is poorly informative as regards the classes involved at level T, since all we get is their set T.need without any further indication. Obviously, no compact representation of T can reproduce the details of dependency relations between classes in T.need. But at least, it would be helpful to make explicit which classes are to be instantiated or not, and which classes introduce polymorphism: such information is available from the C-ORD. It is relevant to re-introduce it in order to facilitate the identification of infeasible testing levels (see Section 3.5), as well as the later assignment of test criteria to the remaining levels. This leads us to propose a textual representation of testing levels, written as a list of class names with additional special characters denoting the role of the classes within the testing level. Given a testing level T, the writing conventions identify the following cases:

- The name of each class under test, i.e., which belongs to T.goal, is decorated with the special character '*'.
- A class whose only role in T.need is to be a parent class must not be instantiated. Its name is written between parentheses.
- A class may be both a parent and a server class. If T.type = Dyn, this class introduces polymorphism, which is indicated by the special character '#': aggregation and association relations may involve instances of this class and/or its children. If T.type = Sta, these relations are tested only through instances of the (parent) server class according to static dependency. This latter case is indicated by the special character '##'.

Using these conventions, the graph in Figure 4a is rewritten as shown in Figure 4b. For example \((\{C\}, \{C\}, \text{Sta})\) has textual representation \(C##(D)\): this is so because C is the goal of this testing level, and we know from the C-ORD that the only role of D is to be a parent class of C. In the same way, \((\{F\}, \{C,D,F,G,H\}, \text{Dyn})\) has textual representation \(C,F,G,H##\) which makes it possible to see at a glance that: (i) F is the target of this testing level; (ii) C and D are parent classes not to be instantiated; (iii) G introduces polymorphism.

### 3.5 Accounting for Abstract Classes

Since the writing conventions explicitly show which classes are to be instantiated or not, we have now to consider the special situation of abstract classes which cannot be instantiated. Accounting for abstract classes, the test order graph is modified in the following two ways.

![Fig. 4. Test order graphs for Example 1](attachment:image.png)
First, when the role of an abstract class in a testing level implies its instantiation, the testing level is infeasible and has to be removed from the test order graph. This is done by deleting the corresponding vertex and merging its incoming and outgoing edges. Removing a testing level does not mean that the class(es) under test at this level are no longer tested. Their test is postponed to the next feasible testing levels (special character ‘#’ is added).

Second, an abstract class in a testing level may be both a parent and a server class (special characters ‘+’ and ‘*’). In this situation, the abstract class loses its role of server class. The server relationship will only be instantiated with children of the abstract class. The only role of the abstract class is then to be a parent class: parentheses are added to special characters ‘+’ and ‘*’.

Returning to Example 1, let us assume that classes A and D are abstract classes. Three testing levels – namely, A#, A,B#,C,(D) and D# (see Fig. 4b) – are removed. Then, the writing conventions of three other levels are changed:

- C#, (D) becomes C#,(D)#.
- A+B,C,(D),E# becomes (A#)+,B#,C,(D),E#.
- A*,B#,*,(D),E#,F,G*,H becomes (A*),B#,C*,(D),E#,F,G*,H.

Figure 4c shows the modified test order graph.

3.6 TOONS Tool

The whole approach is implemented in a software prototype called TOONS (Testing level generator for Object-Oriented Software). Figure 5 shows an overview of TOONS. The tool is written in C (about 2 KLOC) and runs on SUN workstations.

TOONS takes as input the list of the static relationships between the classes, as defined in the class diagram of the application, and information about the presence of abstract classes. Then, TOONS automatically performs dependency analysis, defines testing levels, orders them, and produces the test order graph using the writing conventions.

If the tool is used in early development phases, it makes it possible to compare the graphs induced by alternative class architectures, thus facilitating the systematic analysis of design options from the perspective of testing.

4 CASE STUDY

Using TOONS, our approach was applied to an industrial case study from the avionics domain. The studied program is extracted from an R&D prototype that implements an airborne application: the Automatic Dependent Surveillance (ADS). The ADS enables the aircraft system to send automatically aircraft surveillance data (position, altitude of aircraft, ...) to the currently connected Air Traffic Control centers. The analysis and design phases follow the OMT methodology, and the OO programming language used is C++. The class architecture consists of eighteen classes (see the corresponding ORD in Fig. 6), class 5 being an abstract class.

This ORD exhibits simple static dependencies. But, when considering dynamic dependencies, the testing process becomes much more complex: classes 10 to 18 statically depend on class 1 which is a parent class of classes 2 to 9; then at execution time, classes 10 to 18 may depend on classes 2 to 9. Table 2 shows the results of the dependency analysis for the eighteen classes.

According to these dependencies, 27 testing levels are defined. Then accounting for abstract class 5, one testing level is removed, namely (1), (2), S#. Figure 7 shows the test order graph thus produced by TOONS.
Table 2. Dependency analysis for the ADS case study

![Table image]

5 RELATED WORK

After having presented our method, the tool that implements it, and its application to a case study, it is worth returning to the motivation of our work and its relation to other approaches. The aim is the determination of testing levels for OO programs, with the concern of minimizing the effort to develop stubs. Adopting this concern, the original approach of Kung et al. has been further investigated in two different directions: 1) our work takes place after cycle breaking in the ORD and proposes improvements based on a similar test order; 2) Tai and Daniels [12] examine the problem of cycle breaking, which leads them to propose a different test order. Let us now comment on both contributions.

Keeping in mind the issues raised in Section 2.3, our contribution is best illustrated by a detailed comparison of Figure 1c (resulting from Kung et al.'s approach) and Figures 4b, 4c (resulting from ours). It can be seen that:

- Our representation of testing levels is more informative. In addition to the class under test (Fig. 1c), it shows the other classes involved in the test experiment and how they are involved (Fig. 4b).
- Adopting a graphical display of the precedence relation allows more flexibility in the scheduling of testing levels. Figure 1c requires that C be tested after both A and D, while Figure 4b shows that C is indeed independent of A. This flexibility is important when accounting for the fact that A is an abstract class (Fig. 4c). The removal of infeasible classes is made possible by a graph-based algorithm that could not be used in the case of Kung et al.'s approach.
- Figure 1c indicates that F and H may be tested independently. Figure 4b shows that this is true only as far as static relationships are concerned. When dynamic relationships come into consideration, the testing of F is no longer independent of H.

The introduction of testing levels with type Dyn is an important difference from Kung et al.'s work. This is relevant especially when considering class architectures such as the one in Figure 6, where complexity comes from potential dynamic bindings. The complexity is hidden in Kung et al.'s representation (Fig. 8), while it becomes apparent in ours (Fig. 7). Half of the eighteen classes need a specific dynamic testing level. For classes 13 to 18, these testing
levels correspond to highly integrated subsystems with many possible ways to instantiate the polymorphic links. Tai and Daniels [12] address none of the above issues. Rather, they concentrate on the problem of cycle breaking. They start from a result established by Kung et al.: every directed cycle of an ORD contains at least one association edge. Then, the test order is computed in two steps.

First, a major level number is determined for each class, considering only inheritance and aggregation relationships. Since all association relationships are ignored, there is no cycle and topological sorting can be applied. Note that if class X associates with Y, it is possible to have major_number(X) < major_number(Y). Then a stub for Y is to be developed since X is tested before Y.

Second, classes with the same major level number are further ordered according to a minor level number, based on their mutual association relationships. Here cycles may appear and must be broken in order to apply topological sorting. The algorithm for cycle breaking first considers the removal of edges that would allow stubs to be reused from previous major levels. If this is not sufficient to break all cycles, a finer analysis considers the number of incoming and outgoing association edges for the involved classes, in order to favor edges that break as many cycles as possible.

The authors also propose a procedure to gradually replace stubs by the actual classes as the integration proceeds according to major and minor level numbers.

It is shown in [12] that the test order is different from the one obtained by Kung et al. Yet, both approaches are concerned with the number of stubs. So, how do they compare in this respect? In our opinion, the balance is in favor of Kung et al.'s work for two reasons.

First, the computation of major level numbers in [12] ignores all association relationships, without considering where the cycles are. It is a radical approach that may lead unnecessary stubs to be developed. Let us consider the class architecture for our case study (Fig. 6). Ignoring association relationships will lead to computing major level number 1 for classes 1, 13 and 15. Testing classes 13 and 15 at this first level will necessitate the development of stubs to simulate classes 14 and 16 respectively. Yet, no stub is required by applying simple topological sorting, since the ORD is acyclic. Of course, it may be argued that acyclic ORDs are not the target of Tai and Daniels' work. But the ORD of Figure 6 could be part of a larger architecture involving cycles, so that the point still holds.

Second, this approach relies on the assumption that a stub developed for one class can be reused in any association relationship involving that class. This assumption is highly debatable. A stub simulates only a small subset of the class behavior – otherwise it is no longer a stub but the actual class. Let us assume that classes X and Y both associate with Z. Then a stub for Z whose simplified behavior is adequate from the perspective of X may be inadequate from the perspective of Y. If new stubs are to be developed for each association edge, the test order will not be optimal.

However, the problem of cycles addressed by Tai and Daniels is relevant: further investigation is required to guide the selection of edges for cycle breaking and to determine when to reintroduce the actual classes in place of their stubs. As regards cycle breaking, an interesting direction followed by Jéron et al. [5] consists of adapting classical graph algorithms to minimize the number of stubs. The authors propose a variant of the well-known Bourdoncle's algorithm. Strongly connected components of the dependency graph are first identified. Then the algorithm gradually breaks all cycles by selecting nodes and cutting their incoming edges. The node selection strategy is different from Bourdoncle's one: it is shown on an example that a lower number of stubs is yielded. Since the algorithm does not consider the type of dependencies, inheritance or aggregation edges may be cut, which is undesirable – or even meaningless: one can obtain a test order according to which child classes are tested before their parent class(es), thus requiring stubs to simulate parent classes. But, by introducing appropriate information like the type of dependencies or even the cost of edge removal (if the stub construction effort can be a priori evaluated), further adaptations of the base algorithm are likely to provide efficient solutions to the problem of cycle breaking.

6 CONCLUSION AND FUTURE WORK

This paper presents an approach to define and order the testing levels for OO software. It exploits the class diagram derived from design documents (e.g., using OMT, UML), after cycle breaking. The concise graphical display of the ordered levels we adopt, allows the tester to see at a glance:

1. The general objective of each testing level, expressed in terms of the class(es) that are the target of validation, the subsystem defining an environment for the target class(es), and whether or not testing should exercise dynamic bindings.

2. Which testing levels must be treated in sequence and which ones may be treated independently, in order to minimize the number of stubs to be constructed.

Furthermore, a graph-based algorithm allows us to automatically remove infeasible testing levels due to abstract classes. The whole approach is implemented in a software tool, called TOONS.

Our future work will consider possible modifications of the graph to incorporate additional information on the classes. A first idea is to account for the semantics of the classes under test, to a certain extent. For example, it may be the case that a class is so simple that it acts as a mere driver for another class: considering two distinct testing levels for them may be deemed unnecessary. Another possible modification arises from the reuse of classes developed in the framework of other applications, which may also lead the number of required testing levels to be reduced. These graph modifications should proceed in a way similar to the one proposed for abstract classes: removal (or grouping) of
levels, and adjustments to the textual representation of the remaining levels.

Such an identification and partial ordering of testing levels is a first step toward the definition of an incremental strategy for testing OO programs. In the test order graph, each level is associated with a general objective which has to be refined into precise criteria focusing on the coverage of either intra-class or inter-class features, or both. The choice of criteria should take advantage of what has been tested at previous levels. Here, we can distinguish between two different kinds of integrated subsystems.

For testing levels toward the top of the test order graph, that is, when few classes are involved, solutions have already been proposed in the literature. [4] investigated incremental testing according to the inheritance hierarchy. As in the case of procedural programs, the selection of test data may proceed from either a structural (see e.g., [11, 13]) or a functional (e.g., [3, 14, 15]) viewpoint. For other pointers, the reader may refer to the extensive survey of [2] and to the collection of papers in [9].

Testing of highly integrated subsystems involving many polymorphic server classes sets the most challenging issue: they correspond to testing levels at the bottom of the test order graph. Practical testing methods are still to be defined for them. Note that our representation of testing levels, which displays all the involved classes and shows which ones may introduce polymorphism, should aid in not forgetting potential bindings. It is hoped that, owing to the fact that all static problems and some dynamic ones have been considered at previous levels, it will be possible to manage complexity by following a gradual strategy built on the test order graph.

ACKNOWLEDGEMENTS

Yvan Labiche is currently supported by Aerospatiale Matra Airbus in the framework of the Laboratory for Dependability Engineering (L5). L5 is a Cooperative Laboratory between Aerospatiale Matra Airbus, Electricité de France, Matra Marconi Space, Technicatome, Thomson-CSF and LAAS-CNRS.

We wish to thank Olfa Abbellatif and Luc Pierson very much for their contribution to the implementation of TOONS within the framework of student projects. We are also grateful to the anonymous reviewers for their careful reading and helpful comments.

REFERENCES