Functional Case Test Design to Optimize the Software Development in Italian Tax Processes (Part II): the F23 Tax Payment Form

Fernando Rolli*, Chiara Parretti, Paolo Citti, and Massimo Rinaldi

Department of Innovation and Information Engineering, Guglielmo Marconi University, Via Plinio 44 - 00193 Rome, Italy

Abstract. In this case study, the methodology of test case preparation proposed in the first article is applied to the design of a testing plan in order to verify the compliance of the functional requirements for the compiling system of an F23 tax payment form. The objective of this approach is the definition of the functional requirements coverage matrix to associate one specific test case with each system component. This methodology’s aim is to equip the software designers with an accurate testing plan that is perfectly aligned with the designing logics of the system, respectively in order to optimize the time used for software implementation, to ensure that the tax deadlines are met, and to satisfy the user requirements. This approach also allows to manage a test plan configuration for the software’s future versions, while having, at the same time, an updated technical documentation always available. In fact, with the application of the reuse technique we can identify the software objects affected by the technical and regulatory annual updates. Specific test cases are designed for them, whereas the unmodified components will keep the test cases used in their previous version.

1 Introduction

This article describes the AD methodology application procedures to arrange a functional testing plan process: in this specific case, the implementation involves the compiling procedures of an F23 form using a Holistic non-conformities reduction (HNCR) model [1-3]. The objective of this approach is the definition of the functional requirements coverage matrix to associate one specific test case with each system component. AD is indeed a flexible methodology suitable to be effectively applied to several range of context and scenarios whereby processes [4-5], product [6], product development [7] and teaching approach [8] optimization is required.

1.1 F23 tax payment form

The F23 tax form is used to pay certain types of taxes, fees and penalties, as provided for by the Italian fiscal regulations. More specifically, the F23 model is used to pay taxes such as registration fees, mortgage and land registry fees, which are managed by the Revenue Agency’s offices. It is also used for the payment of fees for public concessions and for the payment of penalties imposed by the judicial and administrative authorities (for example, fines and penalties). In addition, the F23 form can be used by a property owner to collect his/her tenants’ rents. The form is a very efficient tool, as it monitors payment delays, arrears and cash flow. This feature is important in case of disputes between a property owner and a tenant. It also allows to meet the tax obligations established by the Italian law.

1.2 Definitions and correspondences

The Revenue Agency [9, 10] defines and publishes the technical specifics and the F23 model compilation rules, which will be used, in the present study, as customer attributes (CA). For simplicity’s sake, other user requirements are not considered. Starting from this data, the study proceeds with the decomposition of the functional requirements up to the necessary level of detail needed to design the testing plan. Then, it proceeds building the test case coverage matrix (M) in accordance with the proposal described in Part I of the article. Finally, it presents an example of reuse of the test case application, in case of a functionality modification in the F23 payment model. The whole method is based on a zigzagging process between the domain of the functional requirements (FR) and the domain of the data files (DP). This decomposition is summarized by an A correlation matrix. The decomposition process of the functional requirements follows the axioms of independence and information. For the first principle, the A matrix must be a diagonal or triangular matrix, whereas for the second axiom, in
2 Test case preparation

The test case coverage matrix can be built at each decomposition level of the functional requirements. Levels with a major abstraction match with the verifications related to the highest levels of the V model [11].

2.1 Decomposition process

In this context, the most abstract level coincides with the functional system trial test. The test cases that describe the actions and the operating conditions of the program to be tested are explained and made comprehensible to non-technical users. Gradually, as the decomposition of the functional requirements reaches a major level of detail, the test cases are described using technical descriptive languages, thus making them more appropriate for the operating context concerned. The lowest level of functional requirements decomposition is represented by the logical design of the system. At this level, each test case describes the unitary test of each single elementary object. A full definition of the test cases at a higher level can only be produced when the functional decomposition process is over. Therefore, starting from the elementary test cases, coinciding with the unitary test case, we can give a full definition of the test cases at a higher level. This means that the test cases placed at a higher level contain the execution of tests at their lower level. For example, the system test cases are aimed at the execution of the interfaces among the various elementary objects, but they also include the execution of the elementary test. This process leads to the highest level, which corresponds to the procedure’s formal testing. This is illustrated in the V model mentioned in Article I, on the left side of the Figure 1.

2.2 Test case formalization

The test cases represent the specifics of the tests to be executed. They are defined in a theoretical form by the following triplet: < test data input, expected result, application environment and conditions >[12-16]. The set of more test cases constitutes one suite of test cases. Often, in order to be significant, test cases must be performed according to a precise logical sequence, which can be determined by the occurrence of specific conditions. In these situations, a suite of test cases is the entire sequence of tests to be performed in order to verify a specific functional requirement. The test cases must foresee a detailed description of the tests to be executed, so that the test can be repeated by different users and reused in the future versions of the product. This means that the test case must at least contain [11]:

- Test identification code number;
- Description of the test to be executed as a sequence of elementary activities;
- Pre-existing conditions for test application;
- Input data;
- Expected results;
- Anomalies encountered at execution phase;
- Tester identification;
- Date of test execution.

3 Test case decomposition procedure

3.1 First level decomposition

The AD methodology is top-down, therefore the starting point is the highest logical level. In this case, FR = compiling the F23 form. It follows that: FR=F23()*DP, where DP is the data needed to fill in the form. Let the condition be that the F23 procedure to be tested does not consist in a simple editable data entry, but rather that it can be integrated with a more complex system at an enterprise level, where data can be retrieved: this data is available in coded form within the interchange environment with the company’s information system. The F23 payment form to be implemented must be verified by running a test plan [11]. Part of the testing plan consists in the verification of the adherence of the functional requirements specified by the Revenue Agency with the application’s logical design. The specific V model design of the software allows to proceed with the decomposition of the functional requirements and reach the logic implementation of the system’s design that joins the test plan preparation. At this stage, the software’s development process involves the encoding of the system, which is firstly associated with the execution of the test on the developed elementary objects (Unit Test), then with an interface test among the basic components (System Test), and finally with the acceptance test. The trial, instead, is a contractual verification of the adherence of the product to be released with respect to the technical specifications agreed with the supplier. In this context, the test case coverage matrix allows to associate a series of relevant tests to each functional requirement.

3.2 Second level decomposition

3.2.1 Second level decomposition of functional requirements

Proceeding with the decomposition at a lower level, as summarized in Figure 1, we can define, as second level macro functional requirements, both the compilation of the F23’s header and the compilation of the data concerning the actual payment:
The form’s data can be edited or pre-loaded through a specific process, upon request of the company’s information system. In this case, the pre-load process is activated by a specific trigger, which consists of a periodic or an extraordinary event. For example, rent billings or condominium fees allocated to tenants are to be considered as periodic events, whereas the payment of a specific administrative sanction is to be considered as an extraordinary event. In the present analysis, many steps are simplified in order to emphasize the construction of the functional test coverage matrix.

3.2.2 Detection of second level test cases

At this level, we can already start building the test case coverage matrix. The decomposition of the functional requirements enables us to identify, within the correlation matrix, the $A_{ij}$ methods and the $DP_{ij}$ data files that define each functional requirement. In fact, we know that $FR_i = \sum_j A_{ij} DP_{ij}$. If we consider the $A_{11}$ element in Figure 1, we can assume that its relative test case can be formalized in the following data sheet [11]:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Project Name</th>
<th>CT Id</th>
<th>Application Version</th>
<th>Test Case Object</th>
<th>Test Preparation</th>
<th>Test Execution</th>
<th>Input</th>
<th>Expected Result</th>
<th>Tester</th>
<th>Execution Date</th>
<th>Anomalies Found</th>
<th>Test Duration</th>
<th>State</th>
<th>Supervised by</th>
<th>Date of Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>F23</td>
<td>CT1</td>
<td>V1.0</td>
<td>F23 General header compilation</td>
<td>Preparatory conditions list for testing</td>
<td>F23 Header compilation; Payer’s data compilation; Beneficiary’s data compilation;</td>
<td>F23 Header data; Payer’s data; Beneficiary’s data</td>
<td>F23 General header compilation</td>
<td>The F23 general header data must be saved in the DP1 database and must be visualised in the F23 model</td>
<td>XXXXXXXX</td>
<td>03/24/2017</td>
<td>Header insertion not completely insertable</td>
<td>2 minutes</td>
<td>To be executed/In progress</td>
<td>YYYYYYY</td>
<td>03/26/2017</td>
</tr>
</tbody>
</table>

This means that, at this level, the functional decomposition of the user requirements corresponds to the following correlation matrix $A$:

![Fig. 2. Third level correlation matrix A.](image)

In terms of a UML simplified diagram, we obtain the diagram illustrated in the attached Figure 6.

3.3 Third level decomposition

3.3.1 Functional requirements’ third level decomposition

The compilation process of the F23 form is governed by the Revenue Agency’s instructions [9, 10]. Therefore, we propose to match the functional decomposition of our system with the articulation of the sections of the form. At this level, Table 2 shows the correspondence between functional requirements (FR) and data files (DP).

<table>
<thead>
<tr>
<th>Functional requirements</th>
<th>Data files</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR21: fill the header registry</td>
<td>DP11: Header registry</td>
</tr>
<tr>
<td>FR22: compile the payer registry</td>
<td>DP12: Payer’s registry</td>
</tr>
<tr>
<td>FR23: compile the recipient registry</td>
<td>DP13: Recipient’s registry</td>
</tr>
<tr>
<td>FR24: compile the reason for payment</td>
<td>DP21: Type of payment</td>
</tr>
<tr>
<td>FR25: insert the payment purpose codes</td>
<td>DP22: Payment purpose codes</td>
</tr>
<tr>
<td>FR26: compile the amount of payment</td>
<td>DP23: Amount of payment</td>
</tr>
<tr>
<td>FR27: compile the payment details</td>
<td>DP24: Payment details</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional requirements</th>
<th>Data files</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR21: fill the header registry</td>
<td>DP11: Header registry</td>
</tr>
<tr>
<td>FR22: compile the payer registry</td>
<td>DP12: Payer’s registry</td>
</tr>
<tr>
<td>FR23: compile the recipient registry</td>
<td>DP13: Recipient’s registry</td>
</tr>
<tr>
<td>FR24: compile the reason for payment</td>
<td>DP21: Type of payment</td>
</tr>
<tr>
<td>FR25: insert the payment purpose codes</td>
<td>DP22: Payment purpose codes</td>
</tr>
<tr>
<td>FR26: compile the amount of payment</td>
<td>DP23: Amount of payment</td>
</tr>
<tr>
<td>FR27: compile the payment details</td>
<td>DP24: Payment details</td>
</tr>
</tbody>
</table>

The test cases defined at this level are very general: since they entail a very high level of abstraction, they can be considered as functional for the trial test with the client. However, the functional decomposition must be completed before they are wholly formalized. Thus, unlike the functional requirements’ breakdown, which is a top-down process, the decomposition of test cases is a bidirectional process that involves two phases: firstly, a general setting phase which accompanies the functional decomposition; secondly, a phase of methodical definition of the instructions to be followed for each level of decomposition. Therefore, if the decomposition of the functional requirements and the system design are not complete, it is not possible to obtain a comprehensive definition of the testing plan.
2, its corresponding test case can be formalized by the following data sheet:

Table 3. Third level test case formalization.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Project Name</th>
<th>CT Id</th>
<th>Application version</th>
<th>Test Case object</th>
<th>Test Preparation</th>
<th>Test execution</th>
<th>Input</th>
<th>Expected result</th>
<th>Tester</th>
<th>Execution date</th>
<th>Anomalies found</th>
<th>Test duration</th>
<th>State</th>
<th>Supervised by</th>
<th>Date of Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F23</td>
<td>CT11,11</td>
<td>V1.0</td>
<td>F23 Header Compilation</td>
<td>Preparatory conditions list for testing</td>
<td>Execute $A_{11,11}()$, (to fill the header registry)</td>
<td>F23 Header data</td>
<td>The F23 general header data must be saved in the DP11 database and must be visualised in the F23 model</td>
<td>Xxxxxxxx</td>
<td>03/24/2017</td>
<td>Header insertion not completely insertable</td>
<td>2 minutes</td>
<td>To be executed/In progress</td>
<td>YYYYYY</td>
<td>03/26/2017</td>
</tr>
</tbody>
</table>

Therefore, the CT11,11 test case is used to verify the correctness of the $A_{11,11}()$ method related to the DP11 data files. The input data contained in the test case are a significant subset of DP11. Likewise, the other test cases are built for a complete coverage of all the relations present in the relation matrix in Figure 1. Hence, Table 4 enlists, for each method of the matrix in Figure 3, its related associated test case.

Table 4. Matching of the third-level test cases.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Data files (DP)</th>
<th>Related Case</th>
<th>Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{11,11}()$: Header Data Compilation</td>
<td>DP11</td>
<td>CT11,11</td>
<td></td>
</tr>
<tr>
<td>$A_{12,12}()$: Payer’s Data Compilation</td>
<td>DP12</td>
<td>CT12,12</td>
<td></td>
</tr>
<tr>
<td>$A_{13,13}()$: Data Compilation</td>
<td>DP13</td>
<td>CT13,13</td>
<td></td>
</tr>
<tr>
<td>$A_{21,21}()$: Payment Data Compilation</td>
<td>DP21</td>
<td>CT21,21</td>
<td></td>
</tr>
<tr>
<td>$A_{22,22}()$: Tax code no. Pre-Compilation</td>
<td>DP22</td>
<td>CT22,22</td>
<td></td>
</tr>
<tr>
<td>$A_{23,23}()$: Tax code no. Amount Compilation</td>
<td>DP23</td>
<td>CT23,23</td>
<td></td>
</tr>
<tr>
<td>$A_{24,24}()$: Payment Details Compilation</td>
<td>DP24</td>
<td>CT24,24</td>
<td></td>
</tr>
</tbody>
</table>

3.4 Building the functional tests coverage matrix

At this stage, all the elements deemed necessary to build the functional tests case coverage matrix are available. First, each element of Figure 2’s A matrix must contain at least one test case. In this way, we can surely verify all the functional requirements for the system implementation. Unfortunately, a robust-type test case construction is not possible, as it is impossible to build an ideal test that excludes the presence of malfunctions. Tests can detect errors, but cannot exclude their presence [15]. Reporting the test cases in Figure 2’s matrix, we obtain the test case coverage matrix, as described in Figure 3.

Fig. 3. Test case coverage matrix.

3.5 Test case execution mode

The coverage of test cases also carries useful information about the sequence of testing. Considering the test case disposition in Figure 3, we can affirm that almost all test cases can be executed autonomously. In fact, they refer to functional requisites that are logically independent. We may, however, choose a test execution mode that simulates the conventional behavior of the application’s user. In this case, the axiomatic decomposition of functional requirements allowed to obtain the tests execution order, which can proceed from top to bottom, and from left to right. For this reason, the CT11,11 case test can precede CT12,12, CT13,13, CT21,21, CT22,22, and CT23,23. However, test cases related to the functional requisites FR21, FR22, and FR23 are mutually dependent. For the same reason, the CT21,21 case test must precede CT21,22. In this last case, the correlation matrix shows a relation between the two test cases. This relationship of dependence between two test cases might lead to the presumption that the second test execution (CT21,22) is ordinated to the occurrence of a specific condition in the preceding test case (CT21,21). On the other hand, the test cases that are logically independent in the coverage matrix can be only subordinated to the execution sequence, and no logical conditions that can alter the operation are present in the preceding cases. This consideration derives directly from the axiom of independence, because they are arranged along the diagonal of the matrix [17].

4 Test case reuse

The adoption of an axiomatic design allows to reuse not only the software components, but also its supporting documentation. In the specific case of applications designed to manage the tax processing procedures, an annual adjustment is often required in compliance with the regulatory implementations and updates introduced.
by the law. The object-oriented development techniques allow to easily reuse the software components that are not affected by the adjustment measures, to adapt or delete the parts to be modified, and to add new items in order to meet the new functional requirements. In this case, with an axiomatic design approach we can easily identify the test cases to be reused, and to redefine the new matrix of test cases. This implies a reduction in testing time, focusing on the components modified or introduced and their relation with any pre-existing objects. In other words, the axiomatic approach can facilitate a regression testing process on a system that has been affected by an adjustment [11, 14, 16].

### 4.1 Cataloguing the system’s logic components for reuse

The Function Point Analysis is used to identify the logic components that are subject to regulatory changes [18-20]. The function point method allows to estimate the capacity of the software components. This estimate can be used to provide a measure of comparison according to the axiom of information among different functional requirement decompositions. Thus, in the presence of an equal coverage of functional requirements, the most robust solution would be the one with a minor function point size. This would imply less a minor information content. Taking into consideration the purposes of our article, however, the function point technique is used to provide a rational cataloguing of software interventions in terms of regulatory compliance for the years to follow. Cataloguing is a useful tool to optimize the axiomatic process for the preparation of system functional tests. Therefore, the logic components that have been modified are catalogued as follows:

- **DEL.** These are the data files (ILF and EIF) and the transactional functions (EI, EO and EQ) that are deleted, since they are not foreseen by the new functional specifications. The deleting process involves a certain effort, which is equal to the original value in terms of deleted items’ function points, in a percentage reduction;
- **CHG.** These are the data files and the transactional functions that continue to be used because they are still included in the technical specifications. However, they must be adjusted according to the new taxing rules;
- **ADD.** These are, instead, the new data files and the transactional functions provided by the new technical specifications of the Revenue Agency. They were not present in the previous software version. They need implementing.

### 4.2 Software update

According to the Revenue Agency’s resolution dated 17th March 2016, effective from 1st January 2017, taxes, penalties and interests due for the submission of the inheritance declarations must be paid using the “F24” form, as instructed by the Revenue Agency’s resolution n.16, dated 25th March 2016. With reference to the inheritance tax, the F23 form must be used in case this method of payment is specified in the documents issued by the Revenue Agency for the payment of the amounts due mentioned above. These measures required the software vendors to update the compilation procedures. Considering the type of action required, the data contained in the data files had to be updated with the tax codes and the info on the reason of payment. The program structure has remained basically unchanged. The new simplified UML diagram is illustrated in Figure 7, at the last page of this article. This new scheme can be translated into the following test coverage matrix:

<table>
<thead>
<tr>
<th>FR11</th>
<th>CT11, CT12</th>
<th>CT13</th>
<th>CT21</th>
<th>CT22</th>
<th>CT31</th>
<th>CT32</th>
<th>CT33, CT34</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR12</td>
<td>CT21</td>
<td>CT22</td>
<td>CT32</td>
<td>CT33, CT34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR13</td>
<td>CT31</td>
<td>CT32</td>
<td>CT32, CT33, CT34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR23</td>
<td>CT22, CT23</td>
<td>CT23, CT24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR32</td>
<td>CT32, CT33</td>
<td>CT34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FR34</td>
<td>CT34, CT35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 4. Test case coverage matrix with reuse**

This means that, with the exception of test cases affected by the change of tax code and reason for payment, the test plan is unchanged and can be reused for the checks to be performed on the F23 (year 2017 version). As for the three modified test cases, a minimal intervention on the input data file is performed. Some tax codes and reasons for payment have been disabled, and new ones have been inserted. The three test cases are updated, forcing them to follow the new instructions provided by the Revenue Agency. The structure of the coverage matrix corresponds to the decomposition correlation matrix of the functional requirements. This process ensures the complete coverage of the tests and provides useful information on the sequence and the chronological order of their execution.

### 5 Final considerations

The decomposition of the suite of test cases follows the decomposition of the functional requirements specified in the table above. It reaches a sufficient level of detail to define all the necessary elements for its design. This equals to defining, in the decomposition matrix of the testing domain, a suite of test cases for each functional requirement. The obtained test case decomposition matrix mirrors the functional requirements decomposition matrix. It represents the minimum coverage of the test cases that verify all the functional requirements of the system. This solution will never be robust, because, as shown by Dijkstra [15], program testings can detect the presence of malfunctions, but
cannot demonstrate their absence. This means that the functional test coverage does not guarantee the absence of malfunctions in the system. In terms of fiscal management software, our advantage lies in the fact that the fiscal procedures have an annual characterization. Therefore, the observation, the cataloguing and the analysis of historical non-conformities allows to proactively identify any critical areas, as already described in part I. For this reason, any additional test cases that might be required can be entered in the CT matrix. The test case decomposition matrix would no longer respect the axiom of information because it would add test cases with a greater amount of information, but at the same time with a higher probability of detecting the malfunctions. Of course, a test case that contains multiple test trials requires more resources and more time for its preparation as well as for the following execution phase of the developed software. This leads to a trade-off instance, as to whether to release the application first –implying a high risk of non-conformities for the final customer- or to continue with the testing activities and debugging -thus jeopardizing the final delivery deadlines. For this purpose, Musa and Ackermann [11, 21] have designed the so-called fault removal curve, which is based on the following formula:

\[ V(t) = V_0 (1 - e^{-\frac{t}{\lambda_0 V_0}}) \]  

(1)

where \( V_0 \) represents the number of expected errors. This figure can easily be derived if we consider the historical series of the past fiscal software versions [21-23]. \( \lambda_0 \) represents, however, the number of errors detected during the first day of tests. The curve can also be built manually, by recording the number of detected errors on a daily basis. Ultimately, the error removal curve tells us that, even in the presence of an ongoing program testing process, we will never be sure that all malfunctions are deleted. From a certain stage onwards, we can notice that the benefit of detecting an error is minor than the cost of the resources wasted in terms of man / days needed to locate it. For this reason, a limit to the testing activities must be established: this limit varies depending on several parameters such as the levels of service required by the software client and the maintenance costs encountered during this procedure.

6 Conclusions

With the use of AD designing, the testing and functional verification processes are implemented, allowing the construction of a logically coherent testing plan that is integrated with the design logic of the system. This approach is even more effective in terms of taxation management processes, since it allows obtaining the following advantages:

- Meeting the regulatory deadlines set by the Revenue Agency;
- Reducing non-conformities;
- Determining the minimum set of use cases in order to ensure the full functional coverage of the user requirements. At least one test case is allocated for each functional requirement;
- Pre-arranging the system for any future interventions in terms of lower costs and implementation time, especially in case of future regulatory or organizational changes; Reusing the test cases of software’s previous versions.

References

1. C. Cavallini, A. Giorgetti, P. Citti, A. Meneghin, *Sviluppo di un approccio olistico per l’analisi e la risoluzione delle non conformità*, AISS, 3, 1 (2012);
2. A. Giorgetti, C. Cavallini, A. Ciappi, G. Arcidiacono and P. Citti, *A holistic model for the proactive reduction of non-conformities within new industrial technologies*, IJMERR, 6, 4 (2017);
3. A. Giorgetti, G. Arcidiacono, A. Ciappi, R. Barbieri, and P. Citti, *HNCR model following robust approach*, QREI (to be published);
10. Italian Revenue Agency, *F23 payment form*, (Italian Revenue Agency, 2017);
11. A. Avellone, M. Cislaghi, E. Colonese, *Collaudo e qualità del Software*, (AICQ-CI, Uni Service, 2010);
Fig. 6. Simplified UML diagram before reuse.

Fig. 7. Simplified UML diagram after reuse.

12. J.B. Goodenough, S.L. Gerhart, Toward a theory of test data selection, SIGPLAN Notices, 6, 10 (1975);
13. L. Bougè, A contribution to the theory of program testing, TCS 37, 151-181, (1985);
16. I. Sommerville, Software Engineering (Addison-Wesley, 2009);
17. NP Suh, Axiomatic Design - Advances and Applications, (Oxford University Press, 2001);
18. IFPUG, Function Point Counting Practices Manual, (IFPUG, 1999);
21. J. Musa, and A. Ackerman, Quantifying software validation: When to stop testing, IEEE Software, 6, 3, 19–27 (1989);

* Corresponding author: f.rolli@unimarconi.it