CoWTeSt: A Cost Weighted Test Strategy

F. Basanieri, A. Bertolino, E. Marchetti

Abstract

In this paper we present CoWTeSt (COst Weighted TEst STrategy), an original strategy for selecting and prioritarising test cases. Cowtest supports managers to schedule and make cost estimates of the testing stages since the early phases of development. The derivation of test cases is based on the software analysis and design documentation, and uses the UML-based methodology UIT, Use Interaction Test. We report about the application of the proposed strategy to a real case study with some preliminary results.

1. Introduction

One difficulty in project management is to evaluate the cost or required effort of a planned testing phase; this evaluation might be (and is often) done in terms of number of necessary test cases. In this paper will we present Cowtest (Cost Weighted Test Strategy). The advantage of this method is that it helps to decide which and how many test cases should be executed since the early phases of the software development process, well in advance of the coding phase. Cowtest thus represents a practical help for managers to support test planning and to evaluate the impact of the testing phase on the cost of the final product.

In particular, it is possible to apply the proposed method in two ways: if a certain resource investment is fixed for a testing phase, it helps not only to estimate how many test cases to execute, but more importantly to prioritise them and pick those that are judged more useful to assure quality, stability and to evidence problems of the developed product. On the other hand, if the testing stopping rule is given by the coverage of a fixed percentage of functionalities (e.g., 90% test coverage), Cowtest helps to evaluate from the very first stages the cost of such a target and consequently choose the most suitable test cases, as for the previous case.

In our approach we adopt UML, the Unified Modelling Language, that is the emerging graphical notation to model, document and specify OO systems along all the phases of the software process. In literature there are many studies about using UML for design, but only few regarding its usage for testing [1], [2], [4]. We opportunely decided to base our test strategy on UML, so that its application can descend directly from the analysis of the diagrams already developed for the analysis and design phases. Therefore this strategy can be adopted by industries already using UML with little additional effort.

Cowtest supports the organization of the test cases. To derive them, we use a UML-based methodology, called the Use Interaction Test (UIT) method. This has already been presented in [1].

The paper is organized as follows: in Section 2 we give the basic knowledge required for the application of the UIT methodology. Then in Section 3 we describe the details of the Cowtest strategy and present in Section 4 and 5, respectively, a case study and the results obtained applying the method. The conclusion are drawn in Section 6.

2. Background

2.1. UML diagrams

UML [6], [7] is a graphical modelling language to visualize, specify, design and document all the phases of a software development process. It is rapidly becoming the *de facto* standard notation for analysis and design of object oriented software and systems.

UML is based on several types of diagrams, that are graphs differently describing all the aspects, features and phases of a software product. In the following we mention only the diagrams used in our methodology.

Use Case Diagram (UCD): consists of use cases (UCs), actors and their associations. A use case is the representation of a functionality (a specific use) provided by the system. The actor, that can be a system or a person, is an external user that interacts, through associations, with the system, to realize a specific functional requirement. This diagram can be defined in different phases of software development: in the analysis phase, to develop a preliminary design of functional system requirements; in the design, to describe thoroughly all the specific functionalities of the subsystems involved; in the implementation phase, to define the software architecture and the components of the final product. Moreover, the UCD can describe the system at different levels of abstraction, following the incremental development of the system through subsequent refinements and improvements.

Sequence Diagram (SD): shows a number of objects and the messages passed between them, realizing the functionality described in a use case. The realization of a certain functionality through the dynamic collaborations is called a *scenario* and is described by the set of messages exchanged between objects. Also the SD is used at different levels of abstraction and granularity according to the characteristics of the involved objects.

Class Diagram (CD): shows the static structure of the system through the representation of classes with their attributes and methods. Although UML is a language, and as such it does not impose a specific design procedure, an object in a SD often derives from a class in a CD, because it represents one of its possible instantiation in a particular system behaviour realization, which is a scenario.

2.2. Use Interaction Testing

The contribution of this paper is a general strategy to draw and prioritise between a general "universe" of potential test cases. This strategy interacts with a method that derives the (unorganised) test cases from the UML diagrams. The method to derive the tests is the Use Interaction Test (UIT) methodology, and has already been presented in [1]. UIT is an innovative method to derive integration test cases, at different integration or abstraction levels, exclusively based the UML design diagrams. Therefore the main advantage is that it can be already applied in the early phases of the software process, like analysis and design, without additional formalization effort and only using the already existing diagrams in association with background design information.

The construction of Test Cases in UIT is largely inspired by the Category Partition method [5], that is a well-known method to construct functional tests from the specifications. Its partitioning of the input domain of the function to be tested is a standard approach to functional testing, based on the idea that, for the classes of equivalence defined, one can select one or few tests representative of the whole class behaviour.

In our method we analyse one or more SDs relative to a selected Use Case. The integration testing goal is verifying that objects (components, classes, subsystems) interact correctly to perform the required functionality. These interactions are described by the messages that the objects in a SD exchange with each other and so they are exactly the items

to be tested. Thus, each object inside a SD is considered a Test Unit, in the sense that it can be separately tested and represents a possible use of system. For each of these Test Units we derive the relevant Settings and Interactions Categories that are, respectively, environment parameters (or state variables) and messages coming from other Test Units. In this phase we need the related Class Diagram to capture information about parameters values and messages definition. Then a Test Specification is derived finding for each identified category of a Test Unit all the possible values and constraints. Then, observing the temporal ordering of SD messages, it is possible to find the Message Sequences, i.e., a set of messages exploited by objects to define and elaborate specific functionalities. Inside each Message Sequence, we can find a set of Interactions (messages of this sequence) and Settings (attributes that affect the messages) categories. Finally, an executable Test Case is constructed from a Test Specification, taking each of every possible choices, for each involved category.

We report below an example of one test case, generated from the example we will describe in section 4.

The test case is defined for OrigCallSetupUseCase and represents the first test case we can derive from "OrigCall" SD (see Figure 2). Here it is possible to distinguish the InteractionsCategories (e.g. *receiveQ931Msg* or *doMediaFacility*) defined by messages and SettingsCategories (e.g. *pBuffer, len, connKey*) depicted by all the attributes involved in these messages. The test case can be executed considering one of the possible value, taken from its Test Case Specification, of each Settings Categories involved.

Figure 1: A UIT derived Test Case

TEST CASE 1

receiveQ931Msg(`pBuffer, len, connKey, NEAREND`)

`pBuffer`
`len`
`connKey`
`NEAREND`

Q931Msg elaboration [Li+1.TestCase1..4]

[if (Fast Start or Tunneling) perform Media Restriction]doMediaFacility

elaboration of message FAREND [Li+1.TestCase6..7]

sendQ931Msg(`pBuffer, len, connKey, FAREND`)

`pBuffer`
`len`
`connKey`
`FAREND`

Note that the description of the test cases at this level actually remains abstract and can be considered as the specification of test classes. The real executable test cases will be derived instantiating the involved categories values.

3. Proposed strategy

In Section 2.2 we outlined the method UIT to derive the test cases from the UML diagrams. In this paper we want to provide a wider strategy for selecting among the many test cases found by UIT, according to various possible industrial needs. Knowing both the typology and the number of test cases to execute, since the early analysis or design phases, would really be a good result, both for planning different testing activities and for estimating the effort required. Indeed, without a strategy that can discriminate among the many (thousands, or even millions) possible tests, a test derivation method such as UIT, or analogous methodologies, risks being not helpful and too wasteful.

Therefore here below we describe the identified steps of a systematic method for test cost planning and test cases selection and organization in view of software project cost exigencies and estimated risks, which we called Cowtest. The strategy is described in steps, divided into

two groups (corresponding to Sections 3.1 and 3.2). In the former group, we put those steps necessary to build the basic structure (a weighted tree) used for test derivation; in the second group, we put the steps for test case selection in view of different project exigencies.

3.1. Weighted tree derivation

In this section we describe the necessary steps for preparing the basic structure used to apply Cowtest.

1. UCDs and SDs organization

Starting from the main UCD, describing the system functionalities at a very high level, each UC can contain in turn other Use Cases, since, to obtain a complete system functionality, it is generally necessary to execute several actions realizing lower level functionalities. We assume therefore that this criterion is applied to develop a more detailed specification of system functionalities and organize the UCDs in a sort of hierarchical tree.

For each Use Case at the lowest level of the hierarchy, one or more SDs can then be derived, which describe the objects interactions, and exchanged messages, used to realize the Use Case scenarios. As for the UCs we organize these diagrams in the tree in the appropriate position. An example of the obtained structure so far is in Figure 2.

2. Deduce the critical profile

Considering each level of the resulting tree, we annotate every arc with a value representing in a sense the “importance” of the associated node (be it a UC, or a SD scenario) with respect to the other nodes at the same level. The importance is explicitly expressed assigning a significance weight to the representing node of the tree (see Figure 3). The weight values belong to the [0,1] interval and must be assigned in such a manner that the sum of the weights associated to all the children of one node is equal to 1.

The weight should be as high as critical is the functionality represented by the associated node (UC or SD). Several criteria for assigning the importance factor could be developed, for example, evaluating which are the parts, or the functionalities of the system, that will be more significant with respect to usage frequencies, architectural (hw/sw) roles, development complexity, and so on. Generally the knowledge necessary to assign the proper arc’s weight is present in the industrial realities but oftentimes it is not explicitly formalized. Here therefore we are requiring the effort to express in quantitative terms the intuitions and the information about the peculiarity and importance of the system functionalities that must be developed or integrated, considering that such “weight” should correspondingly affect the testing stage.

3. Test case derivation

From the SD, using the UIT method [1] ,the test cases are identified by a combination of all possible choices of variables (or parameters or environmental states) for all message sequences identified in the diagram, as describe in Section 2.2. As in the previous steps we associate to each SD the derived set of test cases. The resulting structure is therefore a tree, in which the root is the main Use Case diagram and the leaves are the SDs with associated test cases at different integration levels.

3.2. Test cases selection and prioritarisation

We depict in this section the steps necessary for selecting the test cases according to the different project needs. Considering the tree developed following the steps described in the above section, the strategy can be applied at different levels of integration. We introduce the concept of *integration stage*:

- The *first integration stage* is represented by the main UC and the SDs (if any), which are children of this node (hence they are at level 2 of the tree).

- The *i-th integration stage* is represented by the UCs positioned in at *i-th* level of the tree and every SDs, children of these nodes, situated at *i+1*-th level.

We decide to consider in the *i-th* integration stage the SDs at level *i+1,* because they represent the interaction between the different components that realize the functionalities described in the UCs at *i-th* level of the tree.

As previously said, it is first of all necessary to define the integration stage we are interested in. The weights assigned to each node contribute to define a relative importance factor, in terms of how risky is that node and how much effort should we put to test it, for each element belonging to the integration stage considered.

In fact considering every node, from the root down to the elements belonging to the integration stage considered, the product of all the nodes weights on the complete path from the root to this node represents its final weight. The final resulting weight associated with each leaf of the tree becomes therefore an element of discrimination for choosing amongst the tests to execute.

We consider now two different situations that can occur in test case selection. The first is the case in which a certain test budget is available, or a fixed number of test cases must be executed. In such a case, with Cowtest, we derive the most suitable distribution of the test cases among the functionalities developed. Regarding the budget, it could be expressed in terms of effort required, man/hours or simply money available: further budgeting consideration is outside the scope of the present paper, we want just to give an idea of how to use the methodology proposed. In our case we consider that we can select on the basis of the leaves weights the number of test cases feasible with respect to the available budget.

The second situation considered is that a certain percentage of functionalities must be covered. In this case by Cowtest we define which are the functionalities to be prioritarily covered and the minimum number of test cases to develop.

In the next section we describe the application of the Cowtest strategy to test selection in the two situations.

3.2.1. Fixed number of tests

In this section we consider the case in which a fixed number of test cases, *NT,* is planned for the testing phase. Cowtest is therefore useful to find a clever distribution of these *NT* test cases among the different functionalities designed.

For each SD, that represents a leaf of the building tree, we use its relative weight, *nw*, corresponding to the chosen integration stage as previously described, for deriving the number of test cases, *nt*, to select among all those associated to the SD. In particular, as many times the final weight is not an integer value, we use for each SDs the following formula:

$$nt = \lfloor nw * NT + 0.5 \rfloor$$

Therefore it is possible to know the number of test cases to plan for each leaf/SD. Using the set of tests associated to each SD (as derived per the UIT methodology) it is possible to choose the tests to be executed and implement them.

In particular associating to each SD a value representing the effort or the cost (for example in terms of man/hours, required budget) it is also possible to calculate the total amount of effort/cost to be scheduled for the testing phase. It is in fact sufficient to multiply the number of test planned for each SD times an average cost and sum all the obtained values (clearly deciding the average cost of test cases is not banal).

Another orthogonal application view of the proposed method is considering a certain budget, *B,* planned for the testing. If no information is available about the cost of the single test cases belonging to the set associated with a SD, the “relative cost” *b* of this SD can be derived using the following formula:

$$b = nw * B$$

The typology of the tests to execute will be selected among the set of tests associated to the SD according to the budget *b* calculated. Otherwise, if the cost of a test case belonging to the set associated to the SDs is available, then it is possible a more appropriate distribution of the total budget.

3.2.2. Fixed coverage

In this section we consider the case in which a certain percentage of functional test coverage (e.g. 90%) must be reached. In this case the proposed methodology can drive the functional choice, highlighting the most critical system functionalities and properly distributing the test cases. Moreover it is possible to distribute, as explained in the previous section, the amount of budget required for reaching the established coverage.

Following the steps described in the previous section for each SD that represents a leaf of the building tree, we consider its final weight, *nw*, calculated for the chosen integration stage.

Considering the coverage to be reached, *C*, the selection of the functionalities to be tested can be easily derived ordering in decreasing manner the *nw*100* values and summing them, starting from the heaviest one, until *C* is reached. In this manner the test effort is focused on the most critical system functionalities, avoiding to devote important test resources towards those less “important”.

It must be noticed that before using the weight for test selection it is necessary to normalize to 1 the selected leaves weight in such a manner that their sum is equal to 1. In particular setting *NW=C/100*, the resulting final weight, *nwf*, for each selected SDs hence is:

$$nwf = nw / NW$$

As shown in Section 3.2.1, having fixed the number of test cases to execute, it is possible to distribute them using the *nwf* values. In orthogonal manner the weight can be also used to derive the number of test cases to be planned. In fact considering the minimum value among the *nwf*, called for simplicity *nwf_min*; multiplying each *nwf* for the factor *f = 1/nwf_min*, and summing the obtained values, the minimum number of test cases required to reach the prefixed coverage, *C*, can be calculated.

In Section 5 some examples are shown.

4. Case study

We present an example from a case study in which we applied Cowtest to a real project provided by a Telecommunication software developer. We consider here only a little part of the analysed system, named SK. In Figure 2 the main functionalities of SK are shown: network access management, network resources management and call management. We analyse in detail the "Call Management" subsystem, that is the part of system that handles all the phases of a Call, that could be defined as a connection between two terminals (PC, GSM phones, IP phones...).

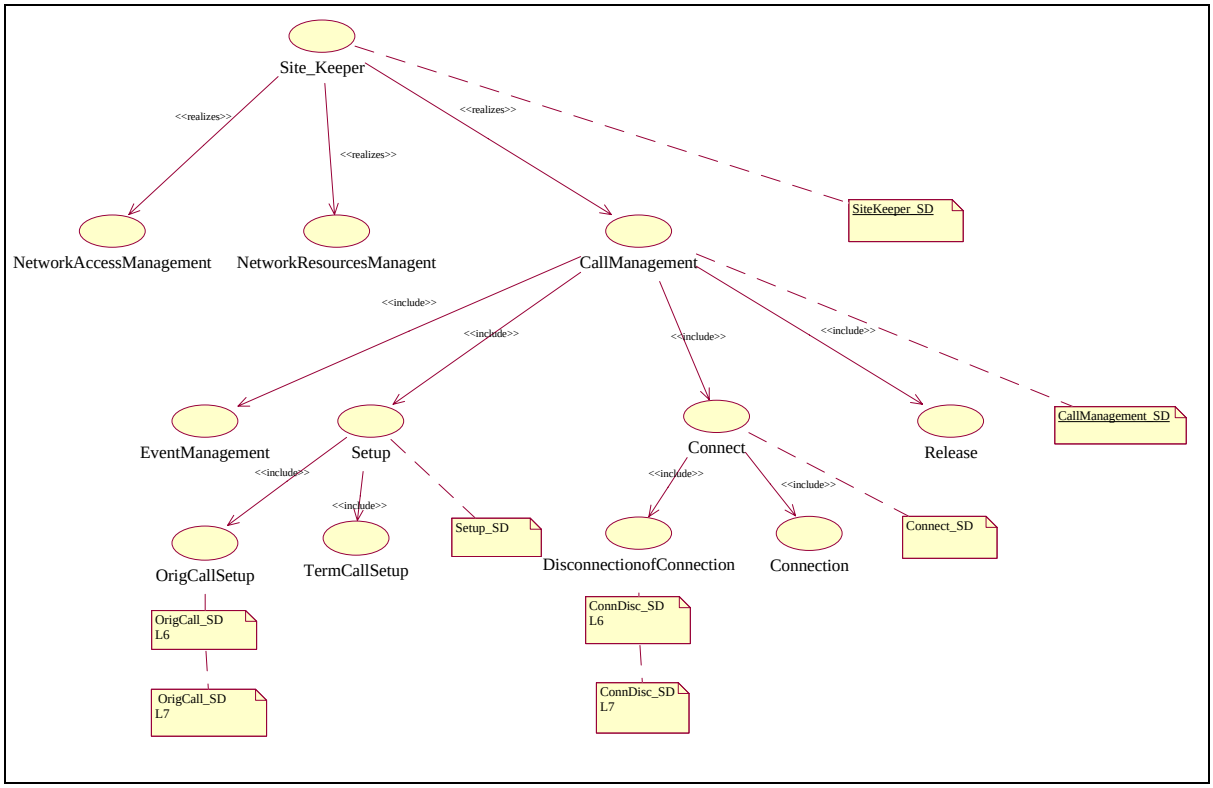

Figure 2: SK main Use Case Diagram

The CallManagement UC can be divided into four sub-UCs:

1 Setup, that is the phase in which caller and receiver are identified and localized,
2 Connect, in which the partners, after their connection, can communicate with each other;
3 Release, in which one of the users involved in the connection decides to terminate the call;
4 Event Management, used to manage all events and signals related to the calls.

Moreover, in the Setup UC the management of a call is divided in two parts: the originating and terminating side, where respectively, the operations and functionalities of caller and receiver are performed. In the same way, the Connection UC is divided in two different behaviours: the connection and the disconnection, describing the two different events that can occur during a call connection phase.

All these functionalities represent the possible call scenarios and so they can be linked to a SD; in Figura 2 we can see the complete tree with all UCs and SDs involved.

A SD describes one of the subsystem functionalities. Each functionality can be realized inside a software component, like modules or packages, or obtained by the interactions of several different components. In the first case we have only a SD, a tree leaf, where objects are the implementation classes of the component that realizes the functionalities. Whereas in the second case, we have more than one levels of SDs such that, the SDs at lowest level are those that realize in detail the described functionality through different actions and the SDs at a higher level represent the integration among all the sub-functionalities to realize the behaviour of the (sub)system as a whole. As shown in Figure 2, there are different levels of integration: SDs linked to the lowest UCs, like "Events management" or "Setup", describe the realization of UCs in detail using specific software components of the system; the SDs at higher levels, like "CallManagement_SD" or "SiteKeeper_SD", are those that describe the interaction among subsystems realizing the main functionalities.

In this manner the test cases can be developed both to verify the integration between different system parts and to investigate on the specific interactions between implementation objects.

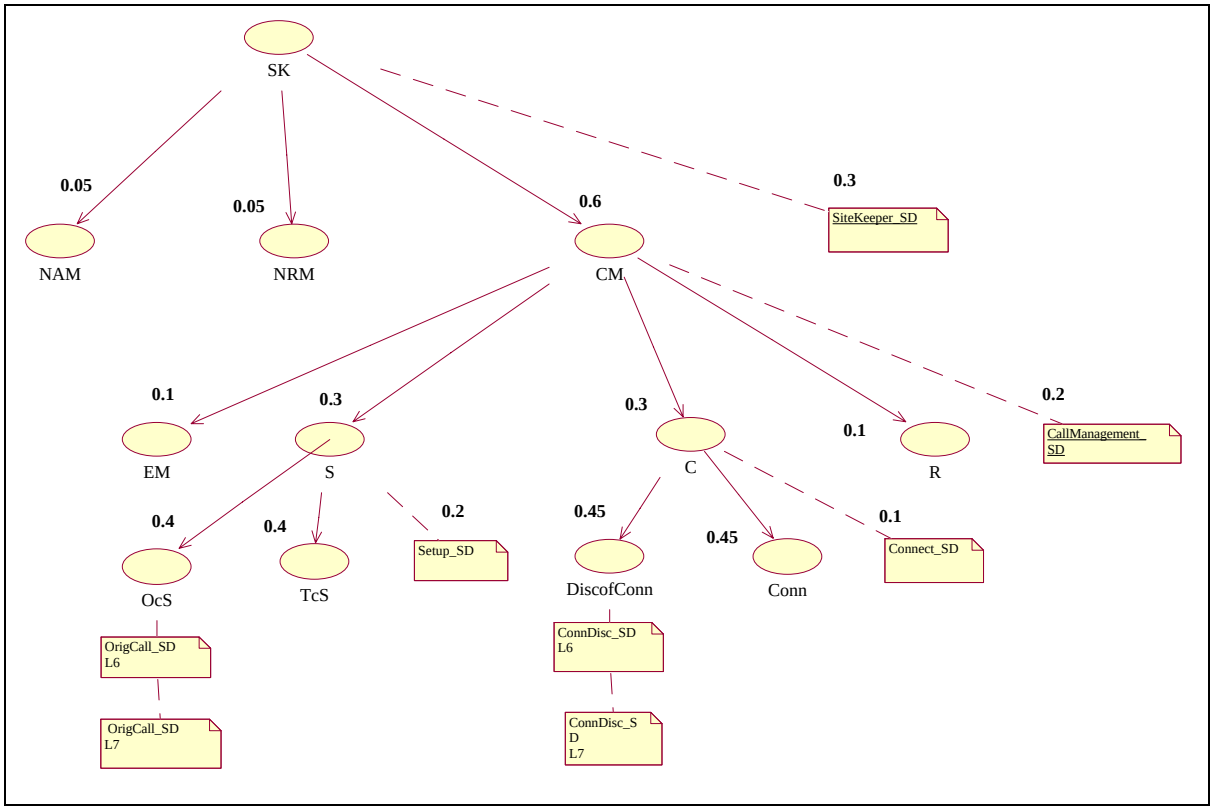

Figure 3: Annotated Cowtest tree

5. Results

In this section we report some results of the presented case study. Applying the methodology described in Section 3, we obtained the annotated tree shown in Figure 3. As already observed, the leaves weights, their critical profile, can be used in different manner depending on the project needs.

We report in Table 1 some results for the situation that the number of test cases is fixed. The weights distribution as well as the number of test case (*NTest*) are redistributed and assigned using the formulas shown in Section 3.

The table is organized in the following manner: the first and the second columns hold respectively the level of integration stage and the name of all tree leaves, that can be UCs or SDs. Note that the names with the suffix *NDchild* (not defined child) are not really present in the tree. They are necessary for representing the part of the tree not yet implemented at the integration stage considered. The third column shows the leaves critical profile. Considering a fixed number of tests to be executed equal to, say, 500, the remaining columns are divided in two parts showing, respectively the relative weight and the final tests number with respect to the selected integration stage.

Note that increasing the integration stage the relative weights and the assigned number of test of each nodes suffixed with NDchild, are redistributed among their children giving a more complete view of the critical functionalities profile.

Int-Stage	Leaves names	Critical profile	2nd Stage/NTest		3rd Stage/NTest		4th stage/NTest	
1st Stage	SK	1						
	SK_SD	0.3	0.3	150	0.3	150	0.3	150
	SK_NDchild	*0.7*	-	-	-	-	-	-

2nd Stage	NAM	0.05	0.05	25	0.05	25	0.05	25
	NRM	0.05	0.05	25	0.05	25	0.05	25
	CM	0.6	-	-	-	-	-	-
	CM_SD	0.2	0.12	60	0.12	60	0.12	60
	CM_NDchild	*0.8*	*48*	*240*	-	-	-	-
3rd Stage	EM	0.1			0.06	30	0.06	30
	S	0.3			-	-	-	-
	S_SD	0.2			0.036	18	0.036	18
	S_NDchild	*0.8*			*0.144*	*72*	-	-
	C	0.3			-	-	-	-
	C_SD	0.10			0.018	9	0.018	9
	C_NDchild	*0.90*			*0.162*	*81*	-	-
	R	0.1			0.06	30	0.06	30
4th Stage	OcS	0.4					0.072	36
	TcS	0.4					0.072	36
	DiscofConn	0.45					0.081	41
	Conn	0.45					0.081	41

Table 1: Distribution of test case at different integration stages

We report in Table 2 some results obtained for the situation that a functional coverage is fixed. We consider the fourth integration stage and several coverage degrees. The weights distribution as well as the number of test case (NTest) are redistributed and assigned using the formulas shown in Section 3.

The table is organized in the following manner: the first and the second columns hold respectively the name of all the tree leaves and the their relative weights at the fourth integration stage. The remaining columns are divided in two parts showing, respectively, the normalized weight and the minimum number of tests with respect to the fixed coverage percentage.

Leaves names	4th Stage weights	70%coverage *nwf* /MinNtest		80%coverage/ *nwf* /MinNtest		90%coverage/ *nwf* /MinNtest		100%coverage/ *nwf* /MinNtest	
SK_SD	0.3	0.413	5	0.354	5	0.137	6	0.3	17
CM_SD	0.12	0.165	2	0.141	2	0.126	3	0.12	7
DiscofConn	0.081	0.111	2	0.95	2	0.085	2	0.081	5
Conn	0.081	0.111	2	0.095	2	0.085	2	0.081	5
OcS	0.072	0.1	1	0.085	2	0.076	2	0.072	4
TcS	0.072	0.1	1	0.085	2	0.076	2	0.072	4
EM	0.06			0.071	1	0.063	2	0.06	4
R	0.06			0.071	1	0.063	2	0.06	4
NAM	0.05					0.052	1	0.05	3
NRM	0.05					0.052	1	0.05	3
S_SD	0.036							0.036	2
C_SD	0.018							0.018	1
Total		72.3%	13	84.6%	17	94.6%	23	100%	59

Table 2: Weights normalization at different coverage percentages

6. Conclusions and future work

In this paper we have presented the Cowtest method and showed its application to a real case study. This example is referred to a simple part of the system with few developed functionalities; therefore the number of possible integration stages is not representative as well as the planned number of tests. However, the obtained results are quite good and encouraging. We are going to validate the complete strategy in an industrial context for the test planning of a complex system with lot of functionalities involved.

Those familiar with Musa's SRET [3] approach should have noticed some similarities between it and Cowtest. In particular, these two methods share a quantitative approach for test cases selection and prioritarisation.

We are developing a tool to automate the Cowtest approach application. In particular, one of the main tool feature will be the integration of Cowtest with a commercial UML design tool so as to allow the smooth integration of the test planning phase with the design stage.

Acknowledgements

The authors wish to thank Ericsson Lab Italy for providing the case study (and the problems to solve!), and more particularly Giovanni Nucera and Gaetano Lombardi for the insightful discussions and nice availability.

References

[1] Basanieri, F., Bertolino, A., “A Practical Approach to UML-based Derivation of Integration Tests”, in QWE2000 conference proceeding, Bruxelles, November 20-24, 3T.

[2] Hartmann, J., Imoberdof, C., Meisenger, M., "UML-Based Integration Testing", in ISSTA 2000 conference proceeding, Portland, Oregon, 22-25 August 2000.

[3] Musa, J,D, “Software-Reliability Engineered Testing”, in QWE1996 conference proceeding, S. Francisco, USA, May 21-24, 2Q2.

[4] Offutt, J., Abdurazik, A., "Generating Test from UML Specifications", in UML conference proceeding , Fort Collins, CO, October 1999.

[5] Ostrand, T.J., Balcer, M.J., “The Category Partition Method for Specifying and Generating Functional Tests”, Communications of ACM, 36(1), June 1998, pp. 676-686.

[6] Rumbaugh J., Jacobson I., Booch J. “The unified Modeling Language Reference Manual”, Addison Wesley, 1999.

[7] UML Documentation version 1.3 Web Site. On-line at http://www.rational.com/uml/resources/documentation/index.jsps/