The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects

Francesca Basanieri, Antonia Bertolino, and Eda Marchetti

Istituto di Elaborazione della Informazione, CNR
Area della Ricerca di Pisa
56100 Pisa, Italy
{f.basanieri,bertolino,e.marchetti}@iei.pi.cnr.it

Abstract. Cow_Suite provides an integrated and practical approach to the strategic generation and planning of UML-based test suites, since the early stages of system analysis and modeling. It consists of two original components working in combination: the Cowtest strategy, which organizes the testing process and helps the manager to select among the many potential test cases, and the UIT method, which performs the automated generation of test cases from the UML diagrams. The approach can be used in incremental way, starting from the preliminary (even incomplete) UML diagrams, and is applied to integration subsystems, as interactively selected by the tester. The emphasis is on usability, in that we use exactly the same UML diagrams developed for analysis and design, without requiring any additional formalism or ad-hoc effort specifically for testing purposes. Cow_Suite has been implemented in a prototype tool, and is currently being validated in an industrial development environment.

1 Introduction

Our research addresses the testing of large, industrial software systems modeled by the UML. The objective of our investigation is to establish an *integrated, practical* and *tool-supported* approach for the *strategic* generation and planning of UML-based test suites, starting since the *early* stages of system analysis and modeling.

To this end, we have developed a methodology and a prototype tool, called **Cow_Suite**, for COWtest pluS UIT Environment. As the name implies, the methodology implemented by Cow_Suite combines two original components: a method to derive the test cases, called **UIT** (Use Interaction Test), and a strategy for test prioritization and selection, called **Cowtest** (Cost Weighted Test Strategy). These two components work in agreement, as Cowtest helps decide which and how many test cases should be planned from within the universe of test cases that UIT could derive for the system under consideration. UIT automatically generates test suites for the high-level test stages, encompassing system and integration testing at various levels. Each generated test suite focuses on a functional portion of the system as

J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML 2002, LNCS 2460, pp. 383-397, 2002.

interactively selected by the tester on a structure of the UML diagrams suitably organized by Cowtest.

Several authors have recently addressed the problem of UML-based testing (as overviewed in Section 4), thus the question naturally arises of which the novel and interesting features of Cow_Suite are with regard to existing methods. Our answer is that the originality of Cow_Suite stems from its emphasis on:

- usability: where other methods require to augment the UML specifications with specific annotations to facilitate the test derivation, or to translate the UML diagrams into an intermediate notation that the methods can process, the leading principle of the Cow_Suite approach is to use for test planning *exactly the same UML diagrams developed for analysis and design*, without requiring any additional formalism or ad-hoc effort specifically for testing purposes. Usability to us means that it is the test methodology that, as far as possible, adapts itself to the modeling notations and procedures in use, and not the vice versa
- timeliness: according to the good software engineering principle that test planning should start as early as possible in the development cycle, a restricted set of minimal preconditions is assumed to start applying Cow_Suite (see Section 2.1). Typically in the early design phases not all relevant scenarios are yet specified and the UML diagrams are defined at a high abstraction level, with several of them yet sketchy. While other methods require a complete and quite detailed set of UML diagrams, Cow_Suite can already start outlining a test plan even at these early stages. Of course the plan will be as abstract as the processed diagrams are and is progressively refined as the diagrams are enriched with more information (see also the incrementality feature below)
- incrementality: Cow_Suite has been conceived for system and integration testing, which are typically conducted in an incremental fashion, considering progressively larger parts of the system and addressing, at each incremental step, the functionalities and the interactions that are relevant at the level considered. In Cow_Suite, the tester interacts with the tool to decide the integration stage for which the test suite should be derived (or, which elements of the UML model should be tested). Then, taking as a reference the corresponding UML diagrams, the UIT method derives the test cases at a specification granularity corresponding to the degree of detail at which the considered diagrams are modeled. We are not aware of other UML-based test methods explicitly addressing incremental testing
- scale: Cow_Suite trades thoroughness for comprehensiveness: as we intend to address UML-based testing of real-world systems in a practical, efficient way, the capability to manage big test suites keeping under control their sizes and functional coverage has been provided via the Cowtest component. Other authors have proposed thorough and meticulous algorithms for deriving detailed test cases, but these methods either cannot scale up to handle the many big UML diagrams that are needed to model huge, complex systems, or would result in an unfeasibly large set of test cases. In contrast, the combined usage of Cowtest and UIT permits to derive a feasible number of test cases while keeping the coverage of functional areas as wide as possible

We have pursued these four features of usability, timeliness, incrementality and scale in organic and systematic manner since the very inception of the approach,

resulting in a UML-based test methodology that is unique and complementary with regard to existing methods.

While preliminary outlines of UIT and of Cowtest have been separately presented elsewhere ([1] and [2], respectively), in this paper we provide an overview of the integrated Cow_Suite methodology (Section 2), including several refinements and improvements to UIT and Cowtest since their early appearance, and describe the Cow_Suite tool in its current status (Section 3). Related work is surveyed in Section 4, and the Conclusions are quickly drawn in Section 5.

2 Cow-Suite Methodology

As already stated, the Cow_Suite approach consists of two components, working in combined way: the Cowtest strategy, described in Section 2.3, and the UIT method, in Section 2.4. Before, in Section 2.1 we express the minimal necessary requirements to Cow_Suite application, and in Section 2.2 we introduce the "Course Registration System" (CRS) [5], to which we refer in the examples.

2.1 Prerequisites to Applying Cow_Suite

The leading criterion of Cow_Suite is to use the same UML diagrams developed during specification and design, without imposing to the UML designers any additional formalism, or ad-hoc effort. The approach can be used in all the phases of the software development process, even though some diagrams have yet to be completed or refined.

Of course, like any other test strategy, Cow_Suite needs to refer to a documented and systematic design process and for this reason we set some minimal requirements. However, they are very basic requirements, in no way test-specific: they establish a minimum discipline in design documentation that should be enforced in any standard software engineering process, and not only for the sake of testers.

Cow_Suite is mainly based on the analysis of the Use Case (UC) Diagrams and Sequence Diagrams[1] (SDs). We need to be able to organize the UCs described in the Use Case Diagrams in a sort of hierarchy, i.e., to explicitly define associations and relations among the developed UCs, and among Actors and UCs, such as, for example, "uses" or "generalization" relationships.

Moreover, it is important to keep trace of how a UC is refined in the low level design; this means to specify how a high level UC, i.e., a system functionality, is realized within the packages of the design model.

Finally, as the UIT method is based on an analysis of the SDs, the description of relevant scenarios results of course essential.

However, in early design phases, it is plausible that the UCs are defined at a high level, and many of them have to be completed, and similarly that not all relevant scenarios are elicited or documented. The Cow_Suite approach can be useful also under these conditions, because it can highlight points of weakness in the reference

[1] Collaboration Diagrams are also usable because, for our purposes, they contain the same information of Sequence Diagrams. Nevertheless, we refer here only to SDs analysis.

documentation, thus providing a picture of the project level of completeness and prompting the user for the revision or the completion of the unfinished diagrams.

2.2 Case Study Description

We consider the Course Registration System (CRS) [5], which is an on-line course registration system for the Wylie College. The CRS users are students, professors and a registrar. They get access to the system via a login function through PC clients. In particular:

- a student can either register to the courses belonging to the current semester course catalogue or view his/her own data relative to the previous semester.
- a professor can select the courses he/she wants to teach from the course catalogue, also defining the dates and times the specific course will be given, and submit the grades.
- a registrar is the responsible of professors and students' information. He/she maintains and verifies the data and the course registrations, checking that there are enough people per course, and notifies the students in case the required courses are cancelled.

The CRS interacts also with a Billing system, which keeps trace of the students registered in each course offering that is not cancelled, so the students can be billed.

In real project development not all system functionalities are developed contemporarily or are specified at the same level of detail. This is the situation we consider: we suppose that the developer concentrates first on the realization of the student system interaction, represented by the system functionalities named: *Login*, used by the students to log into CRS; *View Report Card,* that allows the students to consult their report cards for the previously completed semester; and finally *Register for Courses*, that allows the students to register to courses in the current semester. The *Course Catalog* provides a list of all the course offerings for the current semester, so that the students can also modify or delete previous course selections, if the changes are made within the add/drop period at the beginning of the semester.

2.3 Cowtest

Cowtest (Cost Weighted Test Strategy) provides a practical help to managers for test planning. We provide a stepwise description of Cowtest below. We distinguish two different test planning schemes: testing must respect a certain resource investment, which we translate in practice into fixing the number of test cases; or the test cases must cover a fixed percentage of functionalities. Accordingly, the Cowtest strategy can implement two approaches: fixed number of tests or fixed functional coverage. The choice between either of the two is performed in Step 5.

Step 1: *Identify and organize the graphs representing the design model structure*. A UML design consists of several diagrams made by different model elements, and forming different views of the system. From the main Use Case Diagram onwards, considering each developed diagram and their mutual relationships, we organize the model elements into a defined structure. At this stage we only consider the Actors, UCs, SDs and organize them in an oriented graph MG (V, E), called the *Main Graph*,

representing a global description of the project. The vertices in the set V of MG are the defined model elements[2], and the oriented arcs in the set E represent the relationships between these elements, such as realizes, extends, uses, traces and so on. Figure 1 reports the Main Graph corresponding to the CRS case study.

It may not be always possible to represent the design description with only a single graph. When some connections between the different model elements are missing, or there is some hole in the design, the vertices of the set V result split out into disjoint subsets and the Main Graph is disconnected into more subgraphs.

The model elements can also be organized from a different perspective, by considering the packages and their components. In this case we obtain the graph DG(V', E'), called the *Design Graph*: the set of vertices V' consists of all the developed packages or components and the set of arcs E' represents the dependences between these elements.

The Main Graph and the Design Graph differ for the model elements they consider, but especially for the kind of information they collect. The Main Graph is, in fact, a high level representation of the system: the UCs represent the functionalities or the sub-functionalities of the system and the SDs the description of how the UCs are realized by the interaction between objects and actors. The Design Graph, instead, provides a lower level description of the system: the packages represent the components or sub-components that will be implemented and the graph structure is a mapping of the project architecture.

Due to space limitations, in this paper we only consider the situation in which one single connected graph can be derived for each of the Main Graph and the Design Graph. In particular, the next steps will be developed for the Main Graph only; the procedure for the Design Graph is the same.

Step 2: *Trees derivation.* The Main Graph is explored by using a modified version of the Depth-First Search algorithm [4]. The algorithm produces a forest of several Main Trees. This hierarchical organization constitutes a detailed documentation of what has been developed so far, highlighting the structural decomposition of the functions. More specifically for each of the derived Main Trees:

- The root is always represented by an actor, who is a person (or external system) interacting directly with the system. The actor requests are therefore the functional stimuli to the system.
- The UCs at the first level represent the requirements, each associated with a different functionality the system must realize. In particular, a functionality could be in turn specialized or refined into sub-functionalities, that correspond to the UCs at the second level in the tree.
- The SDs (if any) at the second level of the tree describe the interactions and the exchanged messages among the objects belonging to one of the UCs at first level.
- Considering the *i-th* level of the tree, the UCs represent the description or the realization the sub-functionalities and the SDs the description of the objects interaction of the UCs at the upper level.

[2] In this case we consider in V only the SDs in relation with one or more actors or UCs.

- Some parts of the tree are opportunely marked: they belong to other trees or to repeated nodes and signal the presence in the Main Graph of cycles or of elements reused in more diagrams.

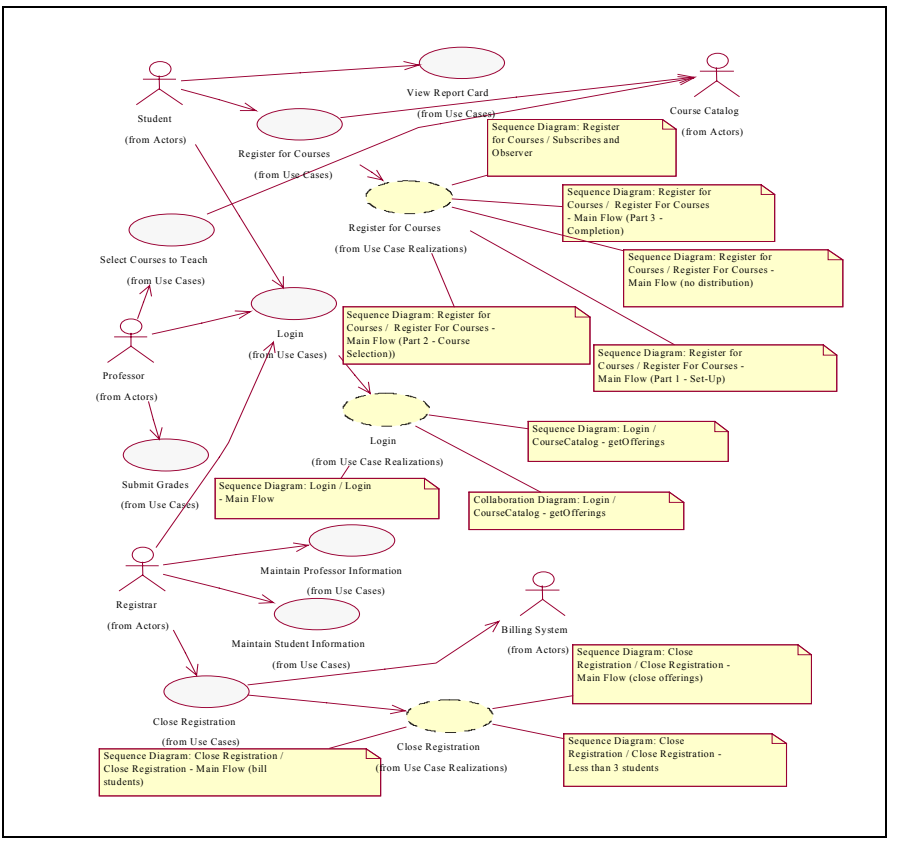

Fig. 1. Main Graph of the Course Registration System

Each level of the tree evidences a different degree of detail of the system functionalities and consequently represents a specific level of integration. Based on the tree levels, we introduce the concept of an *integration stage*, where the *i-th* integration stage is represented by the UCs at *i-th* level of the tree and by every SDs, children of these nodes, situated at *i+1-th* level.

Fig. 5 reports the structure of the Main Tree rooted at the actor Student[3]. The UC node `Login` at first level is filled (and labelled with a "R" not visible in the figure), because it is a multiply used functionality as evident from the Main Graph (more than one actor are associated to it). Considering Fig. 5, the 2nd integration stage is represented by all the SDs at the third level in the tree, the actor `Course Catalog` and the nodes `Login` and `Register for Courses` at second level.

[3] The number in square brackets associated to each node will be explained in step 3.

The trees derivation step can also be applied in the anomalous situation in which the Main Graph is not connected. In this case the modified Depth-First Search produces a set of "anomalous trees", formed by a single model element, or not rooted at an actor, but at a UC. We classify these trees as "Not Linked" and we do not consider them in the strategy application. They will be reconsidered only after the proper associations in the UML design are given.

Step 3: *Assign weights to the nodes*. Generally the various system functionalities do not have the same "importance" for the overall system performance or dependability, and the testing effort should be planned and scheduled consequently. Different criteria can be adopted to define what "importance" means for test purposes, e.g., the component complexity, or the usage frequencies (such as in reliability testing [9]). Oftentimes, these criteria are not documented or even explicitly recognized, but their use is implicitly left to the sensibility and expertise of the managers. Cowtest basic idea is that we ask managers to make explicit these criteria and provide them with a systematic strategy to use such information for test planning.

In particular, considering the derived trees, managers are requested to annotate level by level the nodes with a value, belonging to the [0,1] interval, representing its relative "importance" with respect to the other nodes at the same level. This value, called the *weight*, 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 more critical a node the higher its weight.

Considering the CRS case study, we assumed that `Login` and `Register for Courses` are the new added system functionalities and therefore assigned to them a higher weight than that associated to the already built `View Report Card`. In particular `Register for Courses` is more complex, in term of implemented features, than `Login`, so its testing must be more accurate. Based on these considerations we assign the values 0.50, 0.30, 0.20 to `Register for Courses`, `Login` and `View Report Card`, respectively. In Fig. 5, the weights assigned to each node are represented by the numbers reported in square brackets close to the node name.

Step 4: *Integration stage selection and weighted trees derivation.* Before applying one of the proposed Cowtest strategies, it is necessary to define the integration stage at which the testing is going to be performed. Fixing an integration stage means to decide which nodes to consider for testing, so that the relative weights can be computed. The *final weight* of every node is then computed as the product of the weights of all nodes on the complete path from the root to this node. It is the reference index for choosing among the tests to execute in the next step. Note that the sum of the final weights of the leaves is still equal to one.

For example, in Fig. 5, if the 2nd integration stage is selected, the final weight of the SD `Register For Courses – Main Flow (Part 1 – Set-up)` is 0.1 = 0.25*0.8*0.5.

Step 5: *Cowtest_ing.* The last step of the proposed strategy is to select the method to adopt for test case derivation. As said previously we consider two different situations: either a certain number of tests is fixed, or the percentage of functional coverage is chosen as a stopping rule.

Cowtest_ing with fixed number of tests

If a number NT of test cases is fixed (or, more plausibly, only a test budget up to NT tests can be afforded), our strategy can be used to select NT test cases out of the many test cases that could be conceived. In fact, using the final weight, called *nw*, associated to each SD, the number *nt* of tests to be selected can be easily derived as: $nt = \lfloor nw * NT + 0.5 \rfloor$.

In Fig. 6, for every node of the Main Tree rooted at the Actor `Student`, selecting the 2nd integration stage, and considering a number NT of tests equal to 500, the values NTP represent the number of tests *nt* assigned to every node. For example the assigned number of tests for `Register For Courses – Main Flow (Part 1 – Set-up)` is given by $50 = \lfloor 500*0.1+0.5 \rfloor$.

Cowtest_ing with fixed functional coverage

Let us now consider the alternative case in which a certain percentage of functional test coverage (e.g. 80%) is established as an exit criterion for testing. In this case Cowtest can drive test case selection, by highlighting the most critical system functionalities and properly distributing the test cases.

For each SD that represents a leaf at the chosen integration stage, its final weight, *nw*, is calculated as above. Considering then the fixed coverage C, the selection of the functional test cases to be run can be derived ordering in decreasing manner the *nw**100 values and summing them, starting from the heaviest ones, until C is reached.

For example for the Main Tree rooted at the Actor `Student` (Fig. 5), considering the 2nd integration stage and the final weight of every leaf, the 80% functional coverage is reached covering the nodes: `View Report Card`, `Login- Main Flow`, `Course Catalogue – getOfferings`, `Course Catalog`, `Register for Courses`, `Register For Course – Main Flow (Part 3- Completion)`, `Register For Courses- Main Flow (Part 1 Set-Up)`, `Register For Courses- Main Flow (Part 2 – Course Selection)`. The sum of their final weights times 100 is in fact equal to 81.

Moreover using the final weights of the selected leaves, normalized so that their sum is still equal to 1, it is also possible to derive the minimum number of test cases required to reach the fixed coverage. In this case the minimum number of test cases is 8, one test per leaf except `View Report Card` that requires 2 test cases.

2.4 UIT

UIT, largely inspired by the Category Partition method [11], was originally [1] conceived for integration testing in order to systematically test the interactions among the objects, or objects groups, involved in a SD. Within the Cow_Suite approach, we have integrated a modified version of the UIT method, for clarity called here UIT_sd, by which test derivation is done once for each SD as a whole and not separately considering the objects involved. UIT_sd, similarly to the UIT method, constructs the Test Procedures using solely the information retrieved from the UML diagrams. A Test Procedure, see Fig. 4, consists of a sequence of messages, and of the associated parameters, and instantiates a test case.

UIT_sd is an incremental test methodology; it can be used at diverse levels of design refinement, with a direct correspondence between the level of detail of the scenarios descriptions and the expressiveness of the Test Procedures derived. All the

SDs relative to a selected integration stage constitute the basis for the UIT_sd method. For each selected SD, the algorithm for Test Procedures generation is the following:

1. **Define *Messages_Sequences***. Observing the temporal order of the messages along the vertical dimension of the SD, a Messages_Sequence is defined considering each message with no predecessor association, plus, if any, all the messages belonging to its nested activation bounded from the focus of control region [14]. A Messages_Sequence represents a behavior to be tested and describes the interactions among objects necessary to realize the corresponding functionality.
2. **Analyse possible subcases**: the messages involved in a derived Messages_Sequence may contain some feasibility conditions (e.g., if/else conditions). These conditions are usually described in the message notes or in the message specification and are formally expressed using the OCL notation [14]. If these feasibility conditions exist, a Messages_Sequence is divided in subcases, corresponding to the different possible choices.
3. **Identify *Settings Categories***: for each resulting Messages_Sequence, we define the Settings Categories as the values or data structures that can influence its execution. In detail, they can be determined:
   - from all the messages involved, by considering their input parameters;
   - from the analysis of possible Class Diagrams to which the messages belong, by examining the attributes and data structures that can affect the observed interactions.
4. **Determine *Choices***: for each Settings Category and for each Message belonging to a Messages_Sequence, the possible choices are identified as follows:
   - for the Messages, they represent the list of specific situations, or relevant cases in which the messages can occur;
   - for the Settings Categories, they are the set or range of input data that parameters or data structures can assume.
5. **Determine *Constraints* among choices:** the values of different choices inside a Messages_Sequence may turn out to be either meaningless or even contradictory. To avoid this, the Category Partition methodology suggests to introduce constraints among choices. These are specified by assigning to choices certain *Properties* used to check the compatibility with other choices belonging to a same Messages_Sequence, and by introducing the *IF Selectors*, which are conjunctions of properties previously assigned.
6. **Derive *Test Procedures***: a Test Procedure is automatically generated for every possible combination of choices, for each category and message involved in a Messages_Sequence. For each analysed SD, a document, called the *Test Suite*, collects all the derived meaningful Test Procedures grouped by Messages_Sequences.

Here below, we report an example of UIT_sd application to the SD `Login-Main Flow` in Fig. 2. Following the sequencing of messages along the vertical axis it is possible to initially define (Step1) four Messages_Sequences (M_S) such as:

- M_S1: 1.start(), 1.1.open()
- M_S2: 2.enterUserName(String)
- M_S3: 3.enterPassword(String)

- M_S4: 4.loginUser(), 4.1validateUserIDPassword(String, String), 4.2.setupSecurityContext(), 4.2.1.newUserID(), 4.3.closeLoginSection()

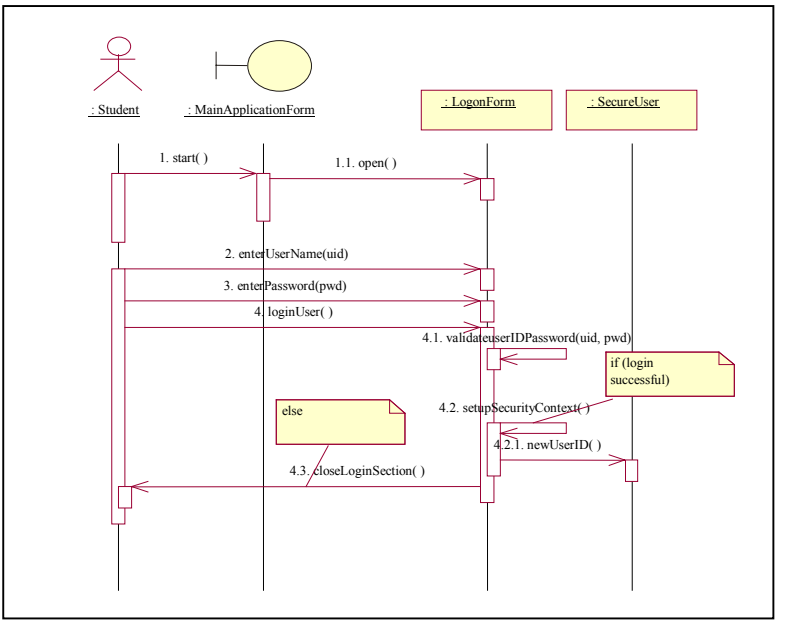

Fig. 2. Sequence Diagram "Login-Main Flow" from CRS example

Choices values for Messages_Sequence 4.1

Settings Categories:

uid
- m.Jackson
- f_smith
- paul_white
- s_71whatson
- ……..

pwd
- m56jkrm
- annamaria
- p71271
- 12.2.73
- …….

Messages:

Loginuser()
- access request of a new user [Property new]
- access request of a registered user [Property registered]
- access request of a not allowed user [Property notAllowed]
- access request of a expired account user[Property expiredAccount]

validateuserIDPassword(uid, pwd)
- access validation of a new user [IF new]
- access validation of a registered user (correct uid and pwd) [IF registered]
- access validation of a registered user (wrong uid or pwd) [IF registered]
- access validation of a not allowe d user [IF notAllowed]
- access validation of a expired account user [IF expiredAccount]

setupSecurityContext()
- successful access of a registered user [IF registered]
- successful access of a new user [IF new]

newUserID()
- access of a new user [IF new]

Fig. 3. Choices values for Messages_Sequence 4.1

As Step 2 describes, a feasibility condition in messages 4.2 and 4.3 can be observed: the value of `login successful` determines the execution of messages 4.2.1 or 4.3 so that Messages_Sequence 4 is split in two different subcases:

- M_S4.1: 4.loginUser(), 4.1.validateIDPassword(String,String), 4.2.setupSecurityContext(), 4.2.1.newUserID()
- M_S4.2: 4.loginUser(), 4.1.validateIDPassword(String, String), 4.3.closeLoginSection()

For each derived Messages_Sequence, the Settings Categories can be identified (Step 3). In M_S4.1, for example, the categories are: `uid` and `pwd`, representing the parameters of the messages involved. Then (Step 4) for each message and for each Settings Category it is necessary to determine the Choices. Fig. 3 shows the definition of Choices for M_S4.1. All the Choices, inserted by the user, are collected in a database and visualized under the Category definition, as shown in Figure 7, so that the user can modify, add or remove them. In Fig. 3, the Constraints values (Step 5) associated to the Choices in square brackets can also be observed. Finally, as described in Step 6, the relevant Test Procedures are generated; the fixed amount of Test Procedures (as imposed by the strategy application) is randomly extracted from the potentially derivable ones. If the Test Procedures to be performed are exceeded, a warning message is issued to the user, who could insert more choices values. Fig. 4 shows one of the derived Test Procedures for the `Login-Main Flow` SD.

Test Procedure
loginUser()
access request of a registered user
validateuserIDPassword(uid, pwd)
access validation of a registered user (correct uid and pwd)
setupSecurityContext()
rID()
access of a new user
uid
f_smith
pwd
m56jkrm

Fig. 4. Test Procedure2 example

3 Cow_Suite Tool

The Cow_Suite approach can be naturally adopted and automated by industries using any UML design tool. We have implemented it into the Cow_Suite tool, designed to be compatible in particular with Rational Rose [13], one of the most widely used commercial tools for UML design. Cow_Suite retrieves the information extracted by Rose from the UML design using the REI (Rational Rose Exthensibility Interface) libraries.

The Cow_Suite tool consists of three working windows: Cowtest, UIT and Test Specification, implementing respectively the Cowtest approach, the methodology UIT_sd and the Test Procedures generation.

The execution starts by analysing the Rose .mdl file (the internal representation of the parsed UML diagrams) and proceeds with the construction of the Main Trees and the Design Tree. Fig. 5 shows the Main Trees, the Design Tree and the list of "Not linked" elements derived for the CRS case study. Looking at this figure, we can notice that the tool provides, continuously as the design evolves, a complete overview of the specification status of the diverse system functionalities.

Considering every Main Tree, the tool, by default, distributes in a uniform way the weights (Step 3) among the nodes at the same level of integration. However the user can always modify any of the assigned weights, and the values of the other nodes are automatically normalized. Following Step 4, the user selects an integration level on a Main Tree and directly chooses, in a dialog window, the test strategy to use. For each selected integration stage, the tool directly derives a weighted subtree according to the chosen test criterion. In Fig. 6, the UIT Tree for the CRS example is reported. In particular the SD nodes keep track of the number of Test Procedures that must be developed accordingly to the test strategy selected. In Fig. 6, the left window shows the selected subtree, while, on the top right, all the SDs are collected together. In the bottom right window only the user selected SDs are listed.

Then, for each selected SDs, the Cow_Suite tool automatically constructs the Messages_Sequences applying UIT_sd. Fig. 7 shows, on the left, the list of Messages_Sequences derived for the CRS example. Each Messages_Sequence contains the lists of all Messages and the Settings Categories involved plus its feasibility condition (where existing).

After Messages_Sequences derivation, the user, using some dialogue windows, can interact with the tool for inserting the Choices values, after which the Test Procedures are automatically derived. As explained in Section 2.4, the tool excludes automatically the combinations of parameters that result contradictory or meaningless. Fig. 7 shows on the right some of the final resulting Test Procedures for the CRS example.

The Test Suite document is so far a text file document, but the Test Procedures final format can be easily adapted to become the input format of a particular Test Driver. To this regard, we remark that the Cow-Suite tool does not execute the derived Test Procedures: to this purpose Cow_Suite should interact with a test driver, to which the derived tests should be passed to be automatically launched.

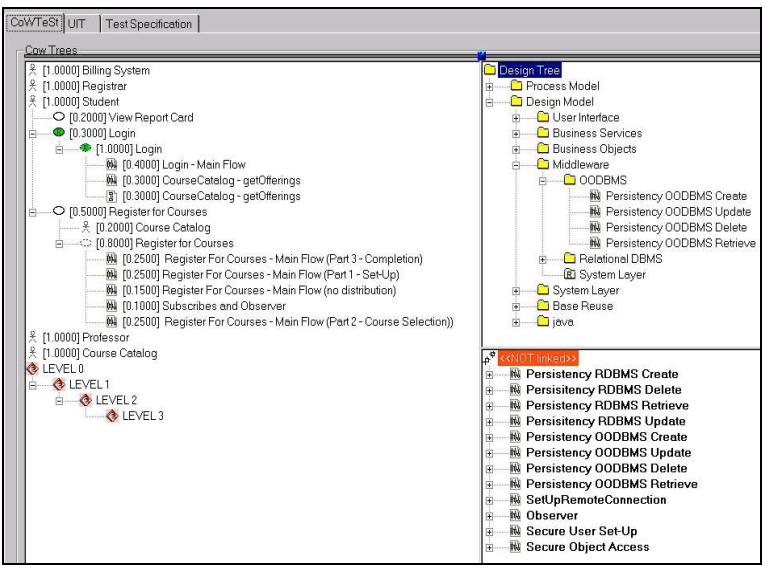

Fig. 5. Main Cow_Suite tool window with Main Trees, Design Trees and Not Linked elements

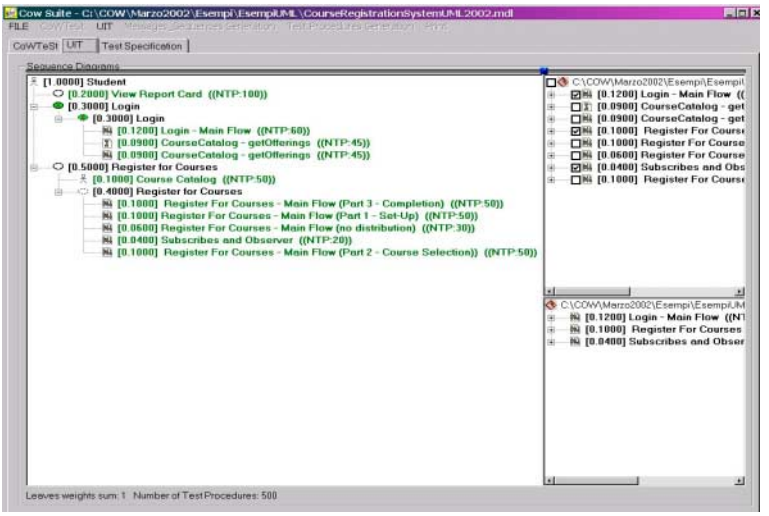

Fig. 6. UIT Window with the derived UIT Tree, the set all SDs found and the selected SDs

4 Overview of Related Work

Even though UML is widely employed in industry and research, only a little part of the literature so far has addressed its use in the testing phases.

In many cases, the tests are derived by translating the UML diagrams into an intermediate formal description, which can be processed by tools already constructed for different methodologies and adapted to the UML specifications. This is the case of [7], in which the authors present a tool, UMLAUT, that is used to transform the UML representation of the system into a form suitable for validation within their VALOODS framework.

In [10] the authors translate the UML State Diagrams into formal SRC specifications, from which input data for unit testing are automatically generated. The same authors have presented in [11] a model for performing static analysis and generating tests inputs from a formal design description of collaboration diagrams specifications. In [8] an approach to derive test cases from UML Statecharts is also investigated, exploiting a formal semantic constructed for UML Statecharts.

Another widespread approach is to augment the UML description with specific notations to support test derivation. This is the case, for example, of the Siemens Corporate Research approach [6], where the developers first define the dynamic behaviour of each system component using a State diagram; the interactions between components are then specified by annotating the State diagrams, and hence the global FSM that corresponds to the integrated system behaviour is used to generate the tests.

In the TOTEM approach [3], the definition of new stereotypes and a rigorous use of OCL notation is required. The authors utilize class invariants and a detailed formal description of UML diagrams (Use Case, Sequence, Collaboration and Class) to early derive test requirements used then to derive test cases, test oracles and test drivers.

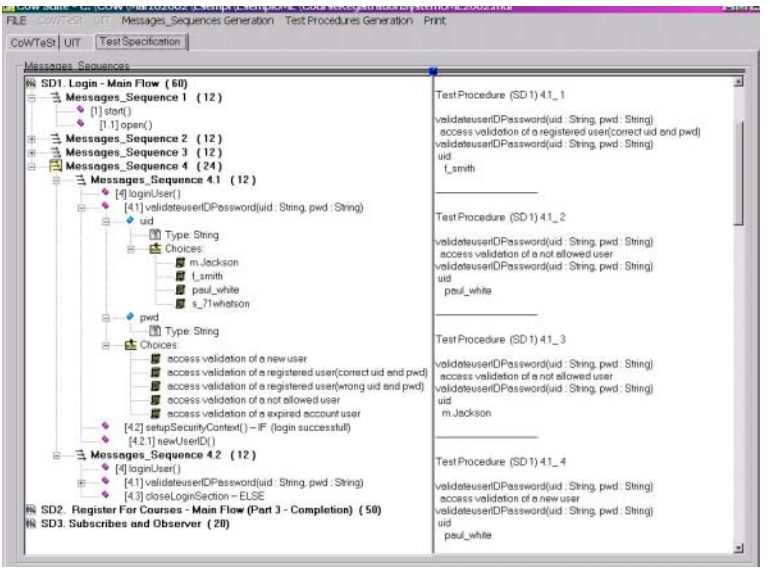

Fig. 7. Messages_Sequences, Choices and Test Procedures for the SD Login-Main Flow

We also mention the recent SCENTOR approach [15], which aims at supporting the generation of scenario-based testing using Junit as a basis.

How Cow_Suite is different from and, in our opinion complementary to, these approaches has been already discussed in the Introduction.

5 Conclusions

This paper contribution is the presentation of the Cow_Suite methodology to support test planning and automated test derivation in UML-based projects. The methodology consists of the combination of the Cowtest strategy for test selection, and of the UIT method for test generation.

The emphasis of our research is in the usability and transferability of the results. We have conceived the Cow_Suite UML-based test methodology keeping in mind the features of complexity, uncertainty and cost characterizing real-world development environments

In this view, we are currently validating the applicability and usefulness of the approach in an industrial context. A comparison between the Cow_Suite derived test plan and the manually derived, standard test plan for a real system, showed that the achieved functional coverage levels were comparable, but the former could be applied earlier and did not require a deep knowledge of the system, as it was the case for the manually derived tests. More experience with the tool is planned.

Acknowledgements

The Cow_Suite project is partially funded by Ericsson Lab Italy (ERI, Rome), in the framework of the PISATEL research laboratory (http://www.iei.pi.cnr.it/ERI). We wish to thank Alberto Ribolini of IEI-CNR, who has implemented great part of the Cow_Suite tool.

References

1. Basanieri, F., Bertolino, A.: A Practical Approach to UML-based Derivation of Integration Tests. Proceeding of QWE2000, Bruxelles, November 20-24, 3T.
2. Basanieri, F., Bertolino, A., Marchetti, E.: CoWTeSt: A Cost Weighted Test Strategy. Proceeding of ESCOM-SCOPE 2001, London, England, 2-4 April 2001.
3. Briand, L.C, Labiche, Y.: A UML-Based Approach to System Testing. UML 2001, Toronto, Canada, 1-5 October 2001.
4. Cormen, T. H., Leiserson, C. E., Rivest R. L., Stein, C. Introduction to Algorithms, Second Edition.The MIT Press and McGraw-Hill, 2001.
5. Course Registration System for Wylie College. On-line at http://www.rational.com/products/rup/resourse_center/examples.jsp
6. Hartmann, J., Imoberdof, C., Meisenger, M.: UML-Based Integration Testing. ISSTA 2000, Portland, August 2000.
7. Jézéquel, J., M, Le Guennec, A., Pennanech, F.:Validating Distributed Software Modeled with UML: Proceeding of UML98, in LNCS 1618, pp. 365-376.
8. Liuying. L., Zhichang, Q.: Test Selection from UML Statecharts. Proceeding of 31st International Conference on Technology of Object-Oriented Language and System, Nanjing, China, 22-25 September 1999.
9. Musa, J.D., Iannino, A., and Okumoto, K.: Software Reliability – Measurement, Prediction, Application. McGraw-Hill, New York, 1987.
10. Offutt, J., Abdurazik, A.: Generating Test from UML Specifications. Proceeding of UML 99, Fort Collins, CO, October 1999.
11. Offutt, J., Abdurazik, A.: Using UML Collaboration Diagrams for Static Checking and Test Generation. UML 2000, University of York, UK, 2-6 October 2000.
12. Ostrand, T., J., Balcer, M.,J: The Category Partition Method For Specifying and Generating Functional Tests". Communication of the ACM, vol. 31, no.6, June 1988, pp. 676-686.
13. Rational Rose tool, On line at http://www.rational.com/products/rose/index.jsp
14. UML Documentation version 1.3 Web Site. On-line at http://www.rational.com/uml/resources/documentation/index.jsps/
15. Wittevrongel, J. Maurer, F.: Using UML to Partially Automate Generation of Scenario-Based Test Drivers. OOIS 2001, Springer, 2001