A Practical approach to UML-based derivation of integration tests

F. Basanieri, A. Bertolino
IEI-CNR, Pisa, Italy

Abstract: *We present an on-going project for developing a tool supported test methodology based on UML descriptions. The leading criteria of this investigation are that no additional language or expertise is required more than UML, and that the methodology can be easily transferred to industrial contexts.*

Keywords:
Category-partition, Integration test strategy, UML, Use-Interaction test.

1. INTRODUCTION

In recent years, the object oriented (OO) paradigm has got widespread use. Software developers need appropriate OO models and support tools for describing and analysing the characteristics of complex distributed systems. OO system models usually involve graphical notations, so that the relationships between the many elements (objects) forming the system can be easily visualized.

UML, the Unified Modeling Language, is the emerging graphical notation to model, document and specify OO systems along all the phases of the software process. Indeed, there exist now many studies about using UML for design, and many tools are available. However, only a little part of these studies so far has addressed the usage of UML for testing, and none of the available commercial tools provide specific assistance for test planning and generation from the UML descriptions.

In this paper we address UML-based testing. Specifically, we want to address this task according to the following two guidelines: (i) we aim at a test method that is entirely based on UML, so that it can be easily adopted by industries already using UML; and, (ii) we refer to high level descriptions of the system: indeed, we want to address test planning for the integration test phase starting from the very first stages of system design.

We present here our practical approach for UML-based integration testing, that is called ***Use-Interaction testing***, as it mainly uses the UML Use Case and Interaction diagrams (specifically, the Sequence diagram). Indeed, integration testing is aimed at verifying that the (hopefully pre-tested) system components interact correctly, and UML Interaction diagrams can provide the information of how the system components should interact.

The proposed approach is very simple, and is inspired at large by the well-known Category Partition method [OB88]: we first look at the Use Case diagram to identify the suitable steps of an incremental test strategy. For each identified sub-Use Case, we then look at the Sequence diagram to identify the relevant components, or "Test Units". For each Test Unit, we derive the relevant "Settings" (could be parameters, variables, environmental states) and the relevant "Interactions" (essentially, messages from other Test Units), and identify for them the significant "Choices" (with the same meaning of the Category Partition method). To do this, we can analyse the related Class Diagrams, as well as other design documentation. Hence, by following the sequences of messages between the components over the Sequence Diagram, we construct the Test Cases, whereby each Test Case is characterized by a combination of all suitable choices of the involved Settings and Interactions.

We have applied the method to a case study, Argo/UML [Argo]. Argo/UML is an open source tool that provide cognitive support to system design. We have developed quite a few Test Cases, and we have been able to identify a few bugs in the case study. However, the method is currently manual, and still under evaluation. Our short term plan is the realization of a tool that support it, by smoothly expanding an existing UML design tool.

The paper is organized in the following way. In the next section, we provide some introductory material to UML and Integration testing. In

Section 3, we describe the method in more detail, explaining it step by step. Then, in Section 4 the case study Argo/UML is briefly introduced, so that in Section 5 some examples of application of Use-Interaction to it are shown. In Section 6 we also provide a very brief sketch of a UML definition of the Use Interaction method itself by means of the UML extension mechanism of stereotypes.

2. BACKGROUND

UML is a graphical modeling language to visualize, specify, design and document all the phases of a software development process. Born by the unification of I. Jacobson, J. Rumbaugh and G. Booch methods, in 1997 it was declared by OMT the standard for analysis and design of Object-Oriented systems.

2.1 UML diagrams

A UML design consists of an integrated meta-model composed of many elements representing OO common world concepts. Through this meta-model we define *Views* showing different aspects of the system to be modelled (Logical View, Component View, Deployment View, Concurrency View and Use Case View). As a whole, these views provide a complete picture of the system to be built. UML diagrams are graphs, each describing the content of a view; they can be arranged in different combinations to provide several system's views.

We only provide in this section a brief description of the UML diagrams mainly used in our methodology (for more details see [UML97a], [UML97b]):

Use Case diagram: a use case is the representation of a functionality (a specific use) provided by the system. This diagram shows a number of external users (actors) and their relationships with the system, when this is used to satisfy a specific functional requirement.

Class diagram: it shows the static structure of the system through the representation of its classes with their attributes and methods. Moreover it specifies the relations between the classes using different types of associations. A system can have more than one Class diagram; in subsequent phases of the software development process, the Class diagram represents objects at different levels of abstraction.

Sequence diagram: it shows the dynamic collaborations between a certain number of objects, highlighting the way in which a particular scenario[1] is realized using the interactions of a (sub)set of these objects. More precisely, a scenario is described by the set of messages exchanged between objects. A Sequence diagram expresses the same information of a Collaboration diagram, whereby the former describes the interactions between the objects during their life-cycle, while the latter shows the relative distribution of these objects links in the space.

2.2 Integration Testing and UML-based Testing

The testing goal is to execute the system to verify its behaviour and to reveal possible failures. Testing is an important piece of the software development process, because of its cost and impact on the reliability of final product. In our methodology we consider the Integration Testing phase, performed to find errors in unit interfaces and to build up the whole structure of software system in a systematic way.

To do this, one could use a non-incremental approach (big-bang), where all the modules are linked together and tested all at once, or preferably, incremental approaches like *top-down*, where modules are integrated from the main program downto the subordinated ones, or, *bottom-up*, where tests are constructed from modules at the lowest hierarchical level and then are linked together upwards, to construct the whole system.

But, when we consider an object-oriented system, the described techniques are not always usable, because, for example, we cannot identify the hierarchical structure of control by which it is possible to define top-down or bottom-up strategies. In an object-oriented environment we can test the class interactions by, for example, integrating together those classes used in reply to a particular input or system event (thread-based testing) or by testing together those classes that contribute to a particular use of the system. In the proposed methodology, the classes to be integration tested (modules, subsystem, processes) are those that realize a system functionality identified from a Use Case diagram.

[1] A scenario is defined as a specific sequence of actions that illustrates behaviour and may be used to show an interaction. [UML97b]

As a matter of fact, even though UML is a powerful mechanism of description, we have found few studies about its use to guide the testing phases.

Some researchers have proposed methods to translate a UML description into another formal description, and then derive the tests from the latter. For instance, in [JGP98] the authors present a tool, UMLAUT, that is used to manipulate the UML representation of the system and automatically transforms it into an intermediate form, suitable for validation. Another interesting paper is [OA99], where a method is proposed to generate test data from UML State diagrams. They translate the UML State diagram into formal SRC specifications, from which input data for unit testing are automatically generated. Finally, in a recent paper from Siemens Corporate Research [HIM00] UML diagrams are used to automatically construct test cases as follows. The developers first define the dynamic behaviour of each system component using a State diagram; the interactions between components are specified by annotating the State diagrams, and then the global FSM that corresponds to the integrated system behavior is used to generate the tests. This approach is being automated to execute the tests in an environment compatible with the UML modelling tool Rational Rose.

All the mentioned studies are interesting, and we see them as complementary to ours, as we do not use State diagrams, but system descriptions at coarser granularity.

3. AN INTEGRATION TEST APPROACH

3.1 Overview of the Use Interaction Testing approach.

As said, Integration Testing progressively verifies the interactions between software components (modules, packages, subsystem, processes, classes) in order to realize the final integrated system.

The ***Use Interaction*** methodology for Integration testing uses as a reference model the UML diagrams to systematically construct and define tests.

The *Use Case diagrams*, by visualizing the various system functionalities, help the tester to decide the way in which the system can be decomposed for testing each of its parts (representing a specific functionality of a Use Case) and then the whole system. Each Use Case 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. In our methodology, Use Case diagrams drive Integration test according to an incremental strategy. We start analysing low-level functionalities that represent a subset of actions described by a sub-Use Case, and then we progressively put them together, until the whole system described in the main Use Case is obtained.

For each selected Use Case we analyse the corresponding *Sequence diagram*, composed by objects and the messages they exchange. The objects involved in the diagram are those that realize and execute the functionality described in the Use Case through elaborations and message exchanges and so they are precisely the components to be tested. In this phase we consider the *Class diagram* too, and particularly a Class diagram at high level of abstraction. This diagram becomes important to define operations (or abstract operations) and attributes required by classes for the interactions of their objects. Also the Collaboration diagram could be used to analyse the object interactions, but it is not as expressive, since it underlines links and dependencies among objects, but not their dynamic interactions along a temporal sequence. Therefore, for our methodology, the most important diagram is Sequence diagram, that is the basis for generating integration tests. In [JBR98], in fact, we can find a suggestion to use this diagram in integration testing phase. They suggest to study different sequences found in this diagram from a possible input state, or from a system input done by actors.

To analyse the message sequences we have used a systematic methodology inspired by the well-known Category Partition Method [OB88].

3.2. The Category Partition method

Category Partition (CP) [OB88] is a well-known method to systematically derive functional tests from the specifications.

Generally speaking, the partitioning of the input domain is a standard approach to functional testing, based on the idea that, for the classes of equivalence defined by the identified partitions, one or few tests can be selected as representative of the whole class behaviour.

The first step of the CP method is to analyse the functional requirements to divide the analysed system in *functional units* to be separately tested. A functional unit can be a high-level function or a procedure of the implemented system. For each defined functional unit, the *environment conditions* (system characteristic of a certain functional unit) and the *parameters* (explicit input of the same unit) relevant for testing must be identified. *Test Cases* are then derived by finding significant values of environment conditions and parameters; this can be done dividing them into *categories* representing relevant system properties or particular characteristics of parameters or environment conditions. Then, for each category, we identify different *choices*, that are different significant values for these categories. The CP method uses a tool to automatically construct *test cases* from specifications expressed into a dedicated semi-formal specification language, called TSL. The CP method has encountered wide interest, and has inspired the development of a large number of test methodologies, also using formal languages such as Z.

3.3 Steps for Use-Interaction testing

We now describe in more detail the proposed approach. It can be logically subdivided into seven steps.

Step 1: *UML Design analysis and search of different Use Cases.* We analyse UML design and, particularly, Use Case diagram. This diagram can lead test construction by defining different integration test stages with respect to an incremental strategy from lower-level abstraction levels upto higher ones. Inside this diagram we can thus find different Use Cases representing (sub)functionalities used to construct the main one (see, for example, Fig.1)

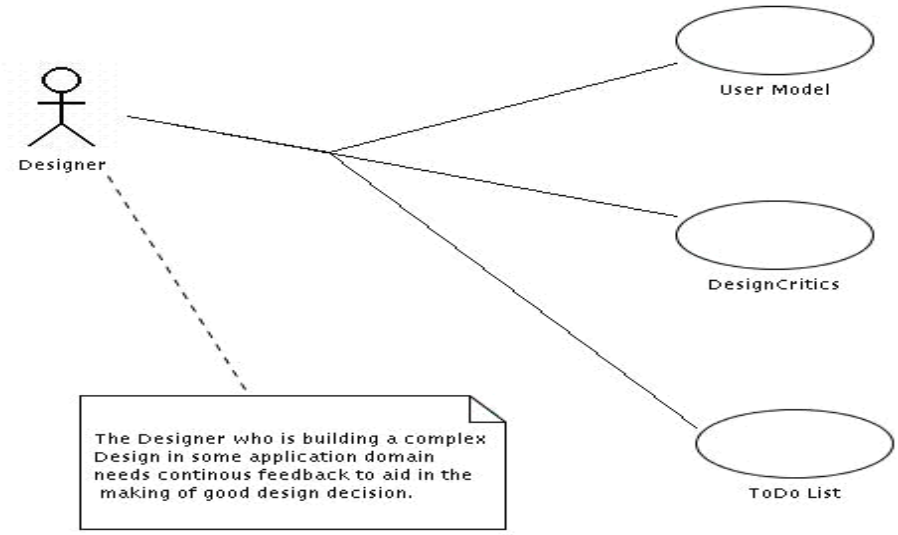


Fig 1: Use Case diagram (of uci.argo.Kernel)

Step 2: *Analysis of Sequence and Class diagrams involved in the selected Use Case.* After selecting one of the Use Cases, we analyse the relative Sequence (Fig. 2) and Class (Fig. 3) (at a high level of abstraction) diagrams. The focus is mainly on the Sequence diagram, analysed along two orthogonal directions corresponding to its axes. The horizontal axis (Arrow a) shows a set of objects that interact through messages; this axis is used to verify the object interactions and their correct use with respect to the Use Case requirements. The vertical axis (Arrow b) is studied because it shows the temporal sequence, from top to bottom along the objects life time, of messages involved in object interactions.

Moreover, horizontal axis is useful for tracing the message sequences, that is at the basis of test cases construction.

Step 3: *Test Units definition.* Each object inside a Sequence diagram is considered a *Test Unit*, since it can be separately tested and it represents and defines a possible use of system. Test Units search can be done automatically because it descends directly from a Sequence diagram.

Step 4: *Research of Settings and Interactions Categories. Interactions categories* are the

interactions that an object has with the others involved in a same Sequence diagram and they are represented by all the messages to the considered object. In this type of diagram, in fact, the interactions are defined with a sender who sends a message to a receiver asking for a service provided by the latter. So, following vertically the life time of the analysed object, we can find all its Interaction Categories represented by the entering arrows. We could also study these methods in the Class diagram description, but it is for searching the *Settings Categories* that the Class diagram becomes very important. Settings categories are attributes (or a subset of them) of a class (and the corresponding Sequence diagram's object), like input parameters used in messages or data structures.

Step 5: *Test Specification construction.* In this step we define a *Test Specification* involved in a Test Unit, that is: for each found category we find all its possible values and constraints. For this aim we use the Class diagram where we can find a preliminary description of a method implementation, its possible input values or the description of an attribute used and its significant values.

Step 6: *Search of Messages Sequences and Test Cases definition.* Following the temporal order of the messages involved in a Sequence diagram, it is possible to find some *Messages Sequences,* i.e., a set of messages used by objects to define and elaborate particular functionalities. Several categories can correspond to each found Message Sequence, of the two types Interaction and Settings: precisely, messages/methods in the considered sequence identify the Interaction Categories, and the attributes that affect these messages identify the Settings Categories. For each category (of either types) within a Message Sequence, we consider each possible choice, taking them from the Test Specification (Step 5). Then, we derive as many *Test Cases* as necessary, by considering all potential combinations of compatible choices. The construction of Test Cases could be performed by a tool, after Messages Sequences and Test Specification have been appropriately defined.

Step 7: *Definition of Use Case Test Suite and Incremental Construction of Test Frame*. Finally, all Test Cases built for a same Use Case are collected together into a *Use Case Test Suite*. If there exist more than one Use Cases to be analysed at the same level of abstraction, we repeat the process from step 2, constructing other Use Case Test Suites.

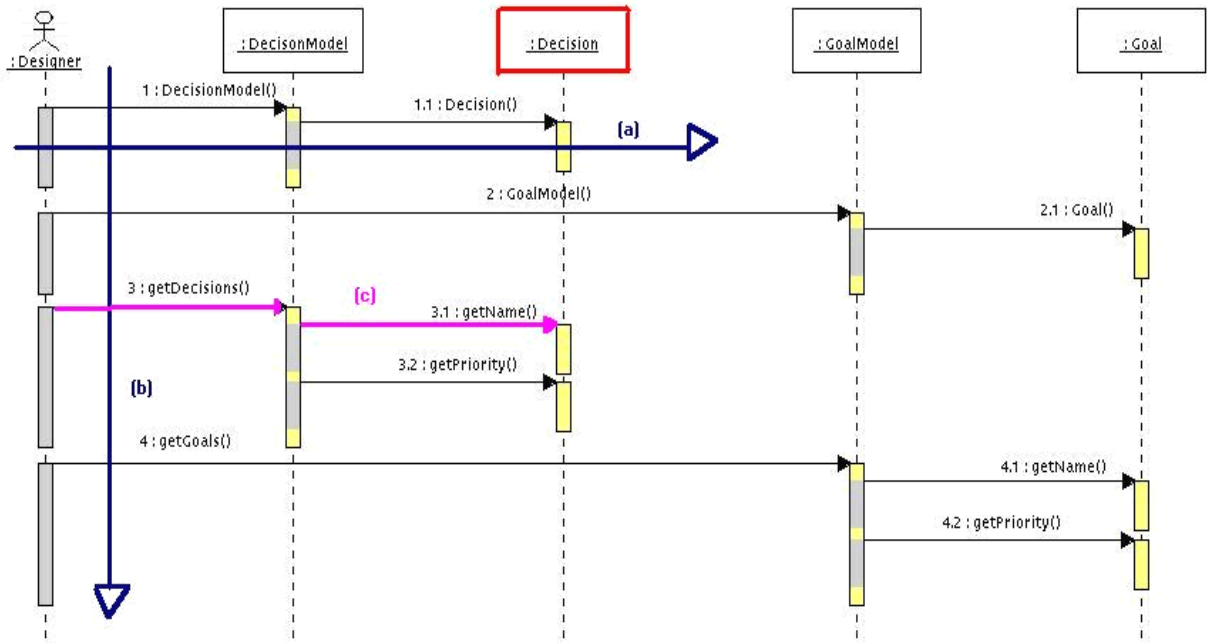


Fig 2: Sequence diagram analysis

After all the (sub)-Use Cases, at a given level, have been analysed, we consider other possible Use Cases at higher levels of abstraction, based on the Use Case diagrams, until we reach the main Use

Case. The same seven-steps process is applied at the identified (higher) Use Cases, using again the Sequence and Class diagrams. As we move towards the main Use Case, the Test Units in the Sequence diagram will generally be more abstract system components than those considered for the (sub)-Use Cases.

Finally, we define a *Test Frame*, that is, the set of all Use Case Test Suites built for the analysed (sub)system.

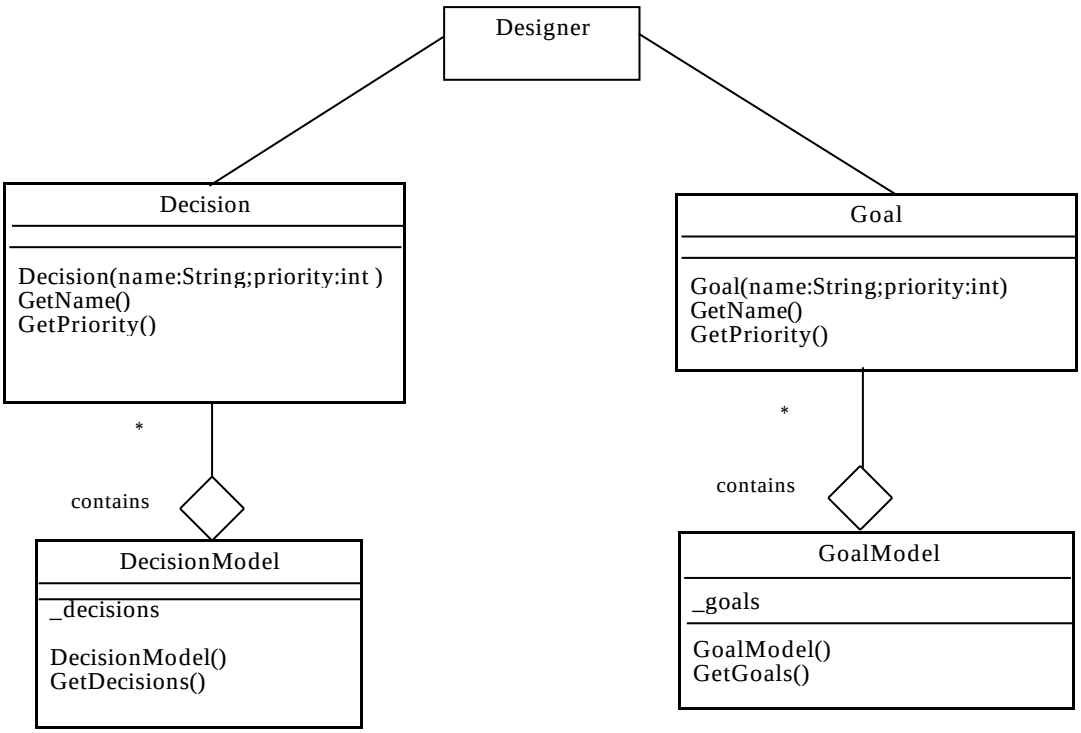


Fig 3: Class diagram of the selected Use case for uci.argo.UserModel

4. THE CASE STUDY

The analysed case study is Argo/UML [Argo], a tool supporting Object Oriented design with UML; this is an open source project that was launched by a research group of the Institute of Computer Science of Irvine, University of California. Argo/UML is based on the UML 1.1 specification and uses a Java version of UML meta-model with support for OCL and XMI (XML Model Interchange format).

We selected this case study because it follows exactly the UML specifications, and it is completely UML designed with good documentation on Java code, even if only a part of its UML design is available on the web. This tool is designed to provide very interesting features, but most of them are not yet developed and only three diagrams are supported so far: Use Case, Class and State Machine diagrams. Its major feature is to provide "cognitive support for design", that is an intelligent support for designers who have to build a complex software product. This support, as the authors describe, is based on: (a) Reflection-in-action; (b) Opportunistic Design.

(a)Reflection-in-action is divided into: Design Critics: simple agents that continuously execute design analysis in a background thread of control, while the designer is working, and suggest possible improvements. The suggestions can be, for instance, the signalling of syntax errors or reminders to return to parts of the design that needed to be finishing; Corrective automations: done by a Wizard for critics identifying specific problems in the design; To Do List: items, organized in a To Do List, helping designer about many details of his work. They are, for example, suggestions from arisen critics, personal notes, suggestion for improving and completing some parts and so on; User Model: maintains information about designer's choices about project and its features. This is done, for example, suggesting only critics that are relevant for designer's tasks. User model consists on a **decision model**, a list of decisions that must be made when doing object-oriented design and **goal model**, a list of goals that designer must reach for the design project. (b) Opportunistic Design divided into: To Do List: a list of To Do Item; CheckList: used to remind designers to cover all design details and avoid

common errors. These lists are specific to the selected design element (association, classes, attributes..).

For our case study we have chosen the part of the tool realizing cognitive support for design, the *Argo* package, divided into two other packages: *uci.argo.Kernel*, that manages Critics, User Model and To Do List and *uci.argo.Checklist*, that manages Checklist. In this paper, for reason of space, we analyse only uci.argo.Kernel. Its functionalities are divided into: **Design Critics**: that manages critics, their definition and use during design. Usable critics are visible in Critic Browser, Fig.4, that keeps track of all information like critic state (active or inactive), priority, general description and so on. All critics properties can be changed by users but it implies changes in To Do items, where you can keep track, every time, of active critics and their priority values. **To Do List**: manages To Do Items, Fig. 5; it shows all active critics descriptions and steps to eliminate them. **User Model**: manages decisions (Decision Model) and goals (Goal Model) made by users for project. **Decision Model**, Fig. 6, is used to specify which requirements are important for designer's aim; they have a priority value from 0 to 5 and if a decision priority is 0 the corresponding critics are made inactive. **Goal Model** shows which goals designer would achieve, Fig. 6. Now only a goal type (Unspecified) is usable; like decisions, goals have a priority from 0 to 5 and, if a goal has priority 0 the related critic is made inactive.

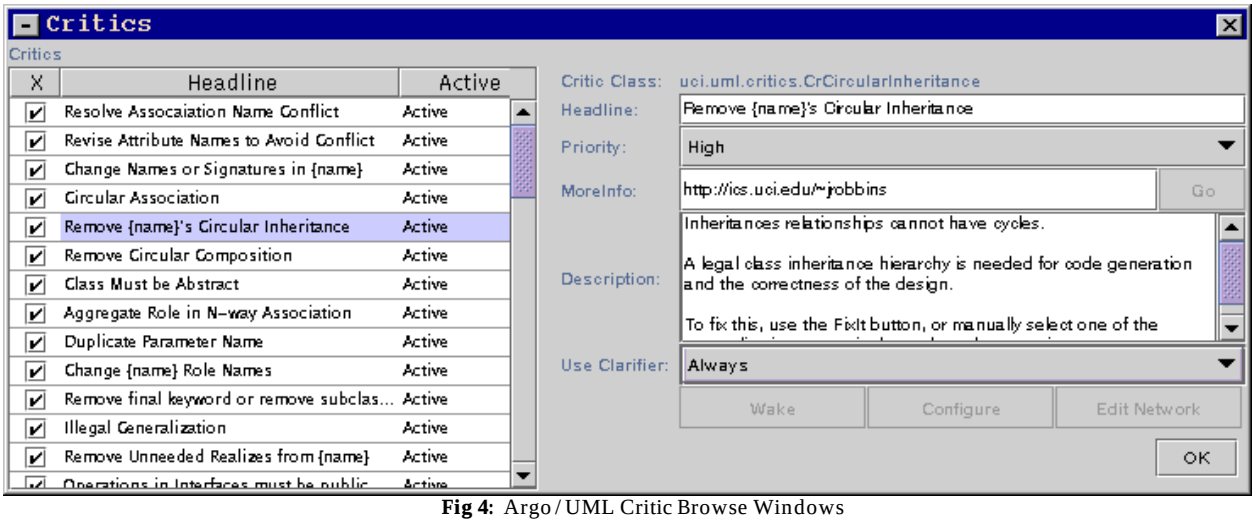


Fig 4: Argo / UML Critic Browse Windows

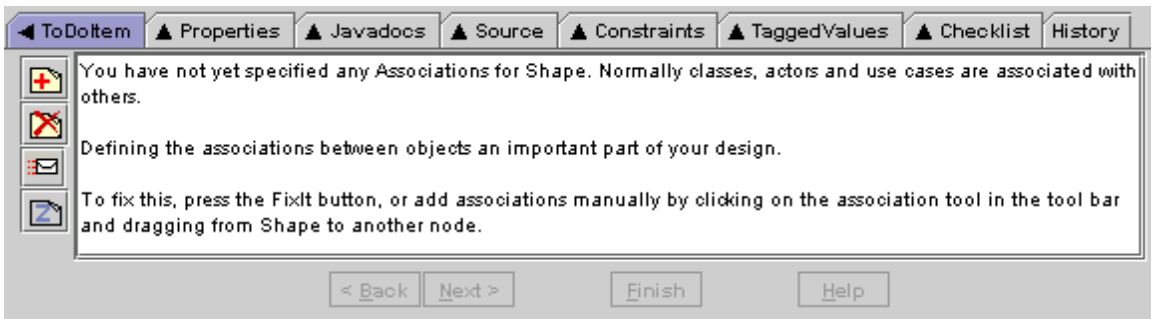


Fig 5: Argo / UML To Do Pane

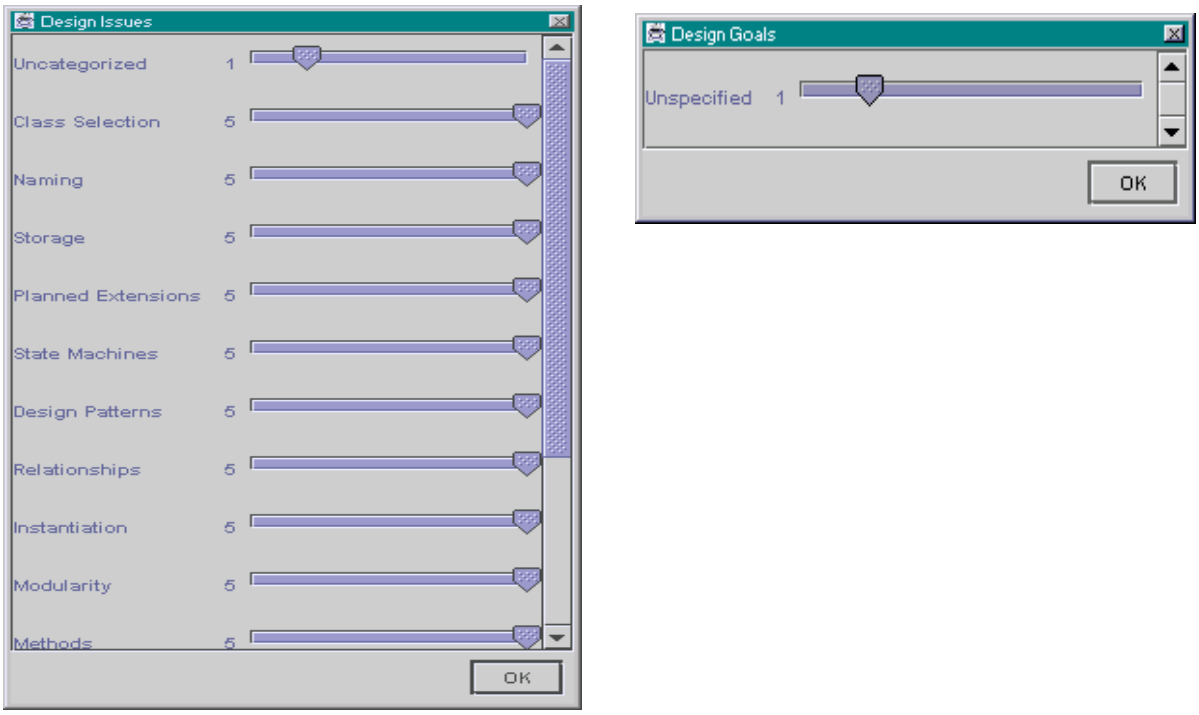


Fig 6: Argo / UML Design Issues Panel and Design Goals Panel

5. APPLICATION OF USE INTERACTION METHODOLOGY TO THE ARGO CASE STUDY

Step 1: *UML Design analysis and search of different Use Cases.* The Use Case diagram of uci.argo.Kernel is shown in Fig. 1. We can see that the main functionality of Kernel is divided into three sub-functionalities, each of which can represent a single sub-Use Case. These Use Cases can be analysed separately to construct the Test Units and Test Cases of a Use Case Test Suite, and then linked together to build one main Test Frame. In this section we analyse only the User Model functionality.

Step 2: *Analysis of Sequence and Class diagrams involved in the selected Use Case.* Fig. 2 and 3 show Sequence and Class diagram for uci.argo.Kernel.UserModel.

Step 3*: Test Units definition.*

TestUnits:

DecisionModel, Decision, GoalModel, Goal.

Step 4: *Research of Settings and Interactions Categories.*

Categories**:**

DecisionModel

setting: _decisions

interactions: DecisionModel(), getDecisions()

Decision

interactions: Decision(name:String,priority:int) ,
GetName(), GetPriority()

GoalModel

settings:_goals

*interactions:*GoalModel(), getGoals()

Goal

interactions: Goal(name:String, priority:int),
GetName(),GetPriority()

Step 5: *Test Specification construction.* We describe the Decision Model Test Unit.

Decision Model: is a part of Design state, it describes which type of decisions are useful for designer; critics relevant for these decisions become active.

Settings: _decision
Naming, Storage, Stereotypes, Inheritance
Relationship, Modularity, ….

Interactions:

GetDecisions()
Opening a new file
Opening a saved file
After a modification and before saving
After a modification and after saving

DecisionModel()
Constructor of class.

Step 6: *Search of Messages Sequences and Test Cases definition.* One of the possible Messages Sequences, highlighted in Fig.2 by Arrow c, is:
getDecisions() -> getName / getPriority(),
and the corresponding Test Case for a possible choice involved categories is:

TEST CASE

getDecisions()

getName()/getPriority()

Opening a saved file.

_decisions = Naming

Action to perform test:Visualizing Design Issues Panel for the considering decision and priority opening a saved document.

Instructions for checking the test: opening the Design Issues Panel we see decision with Name=Naming and priority like priorità of the same file previously saved.

Note 1: This Test must be repeated for all possibile values of _decisions.

Step 7: *Definition of Use Case Test Suite and Incremental Construction of Test Frame.* After defining Test Cases we build a Use Case Test Suite for User Model, Design Critics, To Do List and then, following an incremental integration test strategy, we analyse the main functionality of Kernel constructing its Use Case Test Suite and, at the end, the entire Test Frame.

Results from the case study: when the method has been applied to case study Argo/UML some bugs were discovered. We show two Test Results: the first one, without bugs, is referred to previous Test Case (Step 6); the second is an example of a test with fail result.

TEST CASE 1

getDecisions()

getName()/getPriority()

Opening a saved document.

_decisions = Naming

Test Result: Passed.

TEST CASE 2

SetDecisionPriority (priority:int)

From a >0 value to 0 before saving.

BeInactive()

Critic is active.

RemoveItems(item:ToDoItem)

Critic is disactivated.

RecomputeAllToDoITems()

A decision has priority 0.

_decisions: Naming.

Test Result: Failed.

Fail report: Test is made for all **_decision** values**.** For each of these related critics are disactivated but one critic is never disactived. This is critic 28: "Add operation to <ocl>self</ocl>". This critic is linked to decision *Behavior*, but this type of decision does not exist among built decisions and inside DesignIssuesPanel. Moreover, in the related ToDoPane the critic, even if active, is not visible since it is not linked to any existing decision. The critic can be disactivated only manually from Critic Browse Window.

6. UML DEFINITION OF METHODOLOGY

The Use Interaction testing methodology follows a precise process, and therefore can itself be designed using the UML language. We show, in Fig. 7, a Use Case diagram in which the actor is Tester who interacts with system generating test cases.

Moreover, all the new introduced concepts, like Test Unit or Setting and Interactions Categories, have been defined using exthension mechanism of stereotypes. A stereotype [UML97a] is a new type of modeling element that extends the

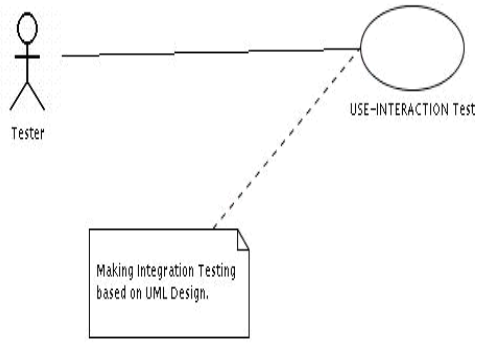
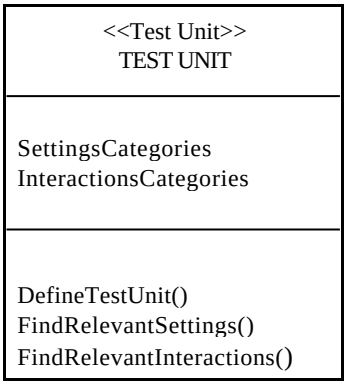


Fig. 7: Use Case diagram of *Use Interaction Test* method.

semantic of the existing metamodel; all our new concepts are (sub)classes of the existing metaclass Class with new additional characteristics and constraints. For instance, we show a stereotyped class, Test Unit, with its properties and constraints defined using the OCL language [UML97c].

Test Unit:



Stereotype Test Unit for instances of meta-class Class

[1] For each Test Unit there exists one and only one related Classifier Role.

self_forAll(t: TestUnit | self.has_exists(c1: ClassifierRole | c1.name = t.name and forAll(c2| c1<>c2 implies c2.name <>t.name)))

[2] The attributes of a TestUnit are SettingsCategories and InteractionsCategories.

self.Attribute.ocltype = enum{SettingsCategories, InteractionsCategories}

[3] A SettingsCategory represents one of the class attributes and it is involved in one iteration.

self.SettingsCategory_exists(a Attribute| a.isused=self and a IsInvolvedIn Interactions)

[4] InteractionsCategories are the only associations of CollaborationDiagram.

self.InteractionsCategory_forAll(i|self.ocltype.Collaboration.Association_exists(ass| ass.name = i))

7. CONCLUSIONS

In this paper we have presented our methodology, *Use Interaction Test*, that generates Integration Tests from UML diagrams like Use Case and Interactions diagrams.

This testing approach is placed inside a larger research project trying to develop methods and tools for design-based integration test of complex system. The approach used in this work is not based on formal methods for specification, like the related [BCIM00], [MLB99] works, but only on exclusive use of UML diagrams.

For this reason, we think it may be a viable method for its simplicity and easy portability to industrial contexts; moreover, since we use only UML diagrams, the methods does not require specialised expertise and analysis, and so generation of test cases can be done contemporary with project development, at no or little extra cost.

On the other hand, the exclusive use of UML diagrams can be considered also a limit of our method, because as known UML semantic is not very precise, and therefore the diagrams can have different interpretations from different users, and also different designers could provide different diagram specifications.

In future, we aim at constructing a tool automatizing part of this process in collaboration with industrial partners. The tool would help the Tester in Test Cases generation for the tasks not requiring human judgement (for example, identification of Test Unit and Messages Sequences), thus enhancing the cost effectiveness of the approach.

Acknowledgements

This work is supported in part by the Italian Murst Project "**Saladin**: *Software Architecture and Languages to coordinate Distributed Mobile Components"*.

References:

[Argo] "Argo/UML", available from http://argouml.tigris.org.

[BCIM00] A. Bertolino, F. Corradini, P. Inverardi and H. Muccini, "Deriving Test Plans from Architectural Descriptions"", *Proc. ACM/IEEE 22nd Int. Conf. on Soft. Eng. (ICSE 2000),* Limerick, 4-11 June 2000, pp 220-229.

[HIM00] J. Hartmann, C. Imoberdof, M. Meisenger, "UML-Based Integration Testing", Proceedings of ISSTA 2000, Portland, Oregon, 22-25 August 2000.

[JBR98] I. Jacobson, G. Booch, J.Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, 1998.

[JGP98] J.M Jézéquel, A. Le Guennec, F. Pennanech, "Validating Distributed Software Modeled with UML", Proc. UML98, in *LNCS 1618,* pp. 365-376.

[MLB99] F. Mercier, P. Le Gall, A. Bertolino, Formalizing integration test strategies for distributed systems", *First Int. ICSE Workshop Testing Distributed Component-Based System,* Los Angeles, May 1999.

[OA99] J. Offutt, A. Abdurazik, "Generating Test from UML Specifications", Second International Conference on the Unified Modeling Language, UML 99, Fort Collins, CO, October 1999.

[OB88] T.J. Ostrand, M.J. Balcer, "The Category Partition Method For Specifying and Generating Functional Tests", Communication of the ACM, 31(6), p. 676-686, June 1988.

[UML97a] UML Notation Guide, v. 1.1, Sept. 1997, *www.rational.com/uml/resources/documentation/formats.jtmpl*

[UML97b] UML Semantics, v. 1.1, Sept. 1997, *www.rational.com/uml/resources/documentation/formats.jtmpl*

[UML97c] Object Constraint Language Specification, version 1.1, Sept. 1997, *www.rational.com/uml/resources/documentation/formats.jtmpl*