Automatic Test Generation: A Use Case Driven Approach

Clémentine Nebut, Franck Fleurey, Yves Le Traon, *Member*, *IEEE*, and Jean-Marc Jézéquel, *Member*, *IEEE*

Abstract—Use cases are believed to be a good basis for system testing. Yet, to automate the test generation process, there is a large gap to bridge between high-level use cases and concrete test cases. We propose a new approach for automating the generation of system test scenarios in the context of object-oriented embedded software, taking into account traceability problems between high-level views and concrete test case execution. Starting from a formalization of the requirements based on use cases extended with contracts, we automatically build a transition system from which we synthesize test cases. Our objective is to cover the system in terms of statement coverage with those generated tests: An empirical evaluation of our approach is given based on this objective and several case studies. We briefly discuss the experimental deployment of our approach in the field at Thalès Airborne Systems.

Index Terms—Use case, test generation, scenarios, contracts, UML.

1 Introduction

It is well known that formal methods can be used both for validating requirements [1], [2] as well as for reducing the cost of testing through automatic test case generation [3], [4], [5], [6]. However, the full formalization of a large object-oriented software for embedded systems also has a cost that many organizations are not ready to pay for. There can be several reasons for that. For instance, at Thalès Airborne Systems (TAS), which is responsible for the inboard software of several combat aircrafts (Mirage 2000-9, Rafale), beyond the problem of finding adequately skilled people, a recurring concern is the lack of integration of formal method with well-established development life cycles. Due to the constant changes in the requirements, this may lead to maintainability problems for the formal specification as well as traceability nightmares. Specifically when dealing with product families, it is economically unrealistic to require a new formalization for each product in the family, or even for successive releases of the same product.

So, instead of pushing formal methods to the industry (one of the mottos in the formal methods community), we propose to work the other way round, i.e., start from established practices and gently lead them toward formally exploitable models. We concentrate here on widely accepted practices based on the use of the Unified Modeling Language (UML) to support an object-oriented development process. We propose a new approach for automating the generation of system test scenarios from use cases in the context of object-oriented embedded software and taking into account traceability problems between high-level views and concrete test case execution.

The method we develop is based on a use case model unraveling the many ambiguities of the requirements written in natural language. We build on UML use cases enhanced with contracts (based on use cases pre and postconditions) as they are defined in [7] or [8]. Lifting up Meyer's Design By Contract [9] idea to the requirement level, we propose to make these contracts executable by writing them in the form of requirement-level logical expressions. Based on those more formalized—but still high-level—requirements, we define a simulation model of the use cases. In this way, once the requirements are written in terms of use cases and contracts, they can be simulated in order to check their consistency and correctness. The simulation model is also used to explicitly build a model of all the valid sequences of use cases, and from it to extract relevant paths using coverage criteria. These paths are called *test objectives*. The test objectives generation from the use cases constitutes the first phase of our approach. The second phase aims at generating test scenarios from these test objectives. In standard development processes [10] based on the UML, each use case is to be documented with several sequence diagrams. Building on these existing sequence diagrams, we automate the test scenarios generation by replacing each use case with a sequence diagram that is compatible in terms of static contract matching. As a result, we obtain test scenarios that are close to the implementation.

The contribution of this paper is thus to generate tests from a formalization of the requirements of a system, in the context of object-oriented embedded software. The goal of the approach is to off-load by automation the work in test generation and to shift the effort to the specification activity. Our objective is to generate test cases for efficiently detecting faults in embedded software. As a starting point, our tests are evaluated in terms of statement coverage. Though the number of covered statements is a very crude measure of test relevance, it is widely used in industrial

- *C. Nebut is with LIRMM (CNRS & Université de Montpellier 2), 161 rue Ada 34392, Montpellier cedex 5, France. E-mail: nebut@lirmm.fr.*
- *F. Fleurey and J.-M. Jézéquel are with IRISA (INRIA & Université de Rennes 1), Campus universitaire de Beaulieu 35042, Rennes cedex France. E-mail: {ffleurey, jezequel}@irisa.fr.*
- *Y. Le Traon is with France Télécom R&D/MAPS, Lannion, France. E-mail: yves.letraon@francetelecom.com.*

Manuscript received 28 June 2005; revised 28 Nov. 2005; accepted 28 Dec. 2005; published online 17 Mar. 2006.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0186-0605.

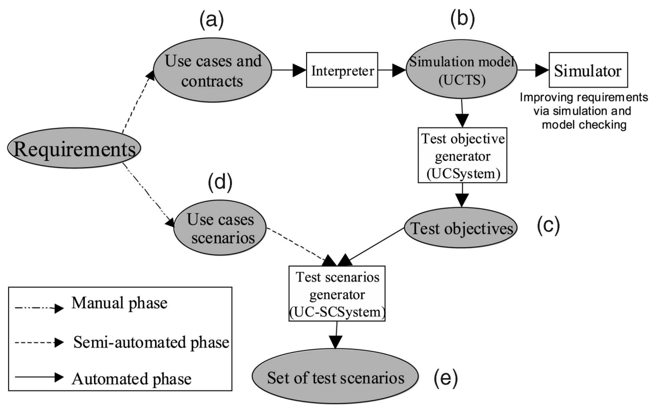


Fig. 1. Global methodology for requirement-based testing.

circles because it is easy to measure even for real-time embedded OO software, where a strict mapping is mandatory from requirements to code.

The remainder of this paper is organized as follows: Section 2 gives an overview of the proposed approach. Section 3 presents the process we use to generate test objectives. It includes definitions for the contract language, the simulation model used, and the criteria to generate test objectives. Section 4 explains how the test objectives are transformed into test scenarios using sequence diagrams. Section 5 presents an empirical validation of the approach by studying the effectiveness of generated test cases on three case studies, in terms of statement coverage. Section 6 discusses related works. We conclude by briefly discussing the experimental deployment of this approach on the field at TAS.

2 Overview

This section provides an overview of the proposed method. Each step is then detailed in the following sections. Fig. 1 summarizes our two-phase method to automatically generate functional test scenarios from requirement artifacts.

The first phase of the method (steps (a) to (c) in Fig. 1), aims at generating test objectives from a use case view of the system. The use cases diagram is a global view describing the system's main functions. Use cases can be seen as high-level functions, as is suggested in the Catalysis method [7]. Yet, these use cases do not correspond to the way these main functions will be implemented in the system. This is why test cases cannot be generated from the knowledge of the use cases alone. However, the use cases may be annotated with pre and postconditions, as proposed in [7]. We thus propose a *requirement-by-contract* approach, inspired by the *design-by-contract* approach of Meyer [9]. These contracts are not executed before and after the execution of a use case as contracts of methods proposed by Meyer, but are used to infer the correct partial ordering of functionalities that the system should offer. Section 3.1 presents a language to express contracts. From the use cases and their contracts, a prototype tool *(UC-System)* builds a simulation model (step (b) of Fig. 1) and generates correct sequences of use cases (step (c) of Fig. 1). In the following, such a correct sequence of use cases is called a *test objective*. As shown in Fig. 1, the use cases model can be simulated. This simulation phase allows the requirement engineer to check and possibly correct the requirements before tests are generated from them.

The second phase (steps (c) to (e) of Fig. 1) aims at generating *test scenarios* from the test objectives. A test scenario may be directly used as an executable test case, or may need some additions from the tester, if some messages or parameters are still missing. To go from the test objectives to the test scenarios, additional information is needed, specifying the exchanges of messages involved between the environment and the system. Such information can be attached to a given use case in the form of several artifacts: sequence diagrams, state machines, or activity diagrams. All these artifacts describe the scenarios that correspond to a use case. For the sake of simplicity, we only deal with sequence diagrams, which are called *use case* scenarios. The principle of the transformation from test objectives to test scenarios is inspired by Briand and Labiche [11] and consists of replacing each use case of the test objective by one of its use case scenarios, using the prototype tool *UC-SCSystem*.

The approach proposed in this paper is designed to be integrated in a classical UML-based software engineering process [10]. The functional requirements are expressed using use cases with contracts for which a dedicated editor has been developed, as explained in Section 3.1. When the main interfaces are designed, the analyst can detail the behavior of each use case using scenarios giving examples of both nominal and exceptional behaviors. These scenarios describe the exact messages that have to be exchanged between the system and the actors. The artifacts required to apply our approach are use cases enhanced with contracts and scenarios attached to these use cases. From these inputs, the generation of test scenarios is automatic. Other approaches [11], [12] also propose to automatically generate test scenarios from use cases and use case scenarios. They are detailed in related work, Section 6. Of course, the quality of the test scenarios strongly depends on the use case contracts and on the scenarios, and the more those artifacts are relevant, the more the generated test scenarios can be relevant.

This two-phase method results in system test scenarios with embeded oracle functions. Several hard points had to be taken into account:

- *Use case and contract validation*: The use cases can be validated through simulation and model-checking. The underlying model has to be compact enough to avoid combinatorial explosion of the internal states of the simulation model, called UCTS (Use Case Transition System). This point is overcome by the two-step approach, which divides the complexity of high-level and detailed requirements into two levels (use cases and sequence diagrams), and by the introduction of use case parameters to deal with main systems concepts and actors.
- *Definition of system test criteria*: Based on the UCTS model, test generation criteria have been proposed that automate the production of test objectives. The most efficient criteria have been identified through experimental comparisons.

This paper is illustrated with the ongoing example of a virtual meeting server. The virtual meeting system offers simplified web conference services. It is used in an advanced software engineering course at the University of Rennes. The virtual meeting server allows work meetings to be organized on a distributed platform. When connected to the server, a user can enter or exit a meeting, ask to speak, eventually speak, or plan new meetings. Each meeting has a manager. The manager is the person who has planned the meeting and has set its main parameters (such as its name, its agenda, etc). Each meeting may also have a moderator, appointed by the meeting manager. The moderator gives the floor to a participant who has asked to speak.

3 Using Contracts on Use Cases to Generate Test objectives

This section presents a technique to express the ordering constraints existing between the use cases of an application, remaining within the UML. This approach proposes to associate contracts—i.e., pre and postconditions—to each use case, in the form of logical expressions.

Such contracts allow the designer to specify both the system properties, making a given use case applicable (precondition), and the properties acquired by the system after its application (postcondition). Use cases as defined by Cockburn [8], for example, or by Catalysis [7], have the notion of pre and postconditions. We propose to make these contracts executable by writing them in the form of requirement-level logical expressions. Because of this executability, our requirement model may be used directly for requirement validation and test case generation. To allow seamless industrial acceptance, the contract language must be simple, so that it can be easily used during the requirements analysis. This declarative approach is very close to the current Thalès Airborne Systems practice.

In the following sections, we introduce the contract language and explain how contracts on the use cases are used for simulation purposes.

3.1 Adding Contracts to the Use Cases

At requirement level, a use case mainly depends on the specific actors to which it is connected, and to the business level concepts it has to handle. Actors involved in a use case can be considered as parameters of this use case. For example, let us consider the use case *open* of the virtual meeting example. It is parameterized by the participant who is opening the meeting and by the meeting to be opened. It is expressed as follows:

$$UC\ open(u : participant; m : meeting),$$

where "participant" and "meeting" are enumerated types. Parameters can be either actors (like the participant) or main concepts of the application (like the meeting). These main concepts, which are identified as business concepts in the requirements analysis, will probably be reified in the design process (i.e., transformed into classes or packages).

When the parameters have been specified for a use case, contracts are expressed in the form of pre and postconditions involving these parameters. The use case contracts are first-order logical expressions combining predicates with logical operators. A predicate has a name, an arity and a set of (potentially empty) typed formal parameters. The predicates are used to describe facts in the system (on actors state, on main concepts states, or on roles).

Since Boolean logic is used, a predicate is either true or false, but never undefined. A system of use cases with pre and postconditions thus needs an initial state that defines which predicates are true at the initial state of the system.

UC open(u:participant;m:meeting)
pre created(m) **and** moderator(u,m) **and not** closed(m) **and not** opened(m) **and** connected(u)
post opened(m)

UC close(u:participant; m:meeting)
pre opened(m) **and** moderator(u,m)
post not opened(m) **and** closed(m) **and**
  forall(v:participant) {**not** entered(v,m) **and**
  not asked(v,m) **and not** speaker(v,m) }

Fig. 2. Contracts of use cases open and close.

Classical Boolean logic operators are used: conjunction (*and*), disjunction (*or*), and negation (*not*). Quantifiers are used to increase the expressiveness of the contracts: these quantifiers are *forall* and *exists.* In addition, an implication operator (*implies*) can be used in the postcondition of a use case to guard an expression with a condition evaluated in the context of its precondition (that must be postfixed with the @pre operator, like in the OCL).

The precondition expression is the guard of the use case execution. The postcondition specifies the new values of the predicates after the execution of the use case. When a postcondition does not explicitly modify a predicate value, it is left unchanged. An example of such contracts is given in Fig. 2. The use case *open* requires that the actor performing the opening on a meeting is its moderator and is connected, and that the meeting has been created, and neither closed nor already opened. After performing the use case *open*, the meeting is opened. The use case *close* requires the meeting to be opened, and the actor performing it to be its moderator. After closing a meeting, it is closed, not opened, and all its participants have left (in particular, nobody is waiting to speak nor speaking). After opening a meeting, an actor can immediately close it, since the precondition and the postondition of this action implies the precondition of the action.

3.1.1 Limitations of the Contractualized Use Case Model

Difficulty of building a contractualized use case model. The declarative definition of such contract expressions forces the requirement analyst to be precise and rigorous in the semantics given to each use case and, thus, may not be so easy to build. The set of used predicates can be seen as the vocabulary to describe the requirements, it is thus necessary to keep the predicate names consistent. Moreover, there may exist dependencies between the predicates, and our approach does not manage those dependencies. Deriving and maintaining a nonredundant, minimal set of contracts and predicates can thus be a complex task, in particular, when the requirements elicitation process involves numerous people and a great number of predicates. To decrease this complexity, we have developed an editor tool to manage the predicates and guide the design of contracts. To check that the requirements are correctly described, we propose the use of a simulator of the requirements (see Section 3.2). Other approaches propose to model the dependencies using graphical notations such as activity diagrams [11], or dependency charts [13], they are discussed in Section 6.

Numeric types. The requirements that can be expressed with contracts on the use cases are high level ones, e.g., they are not suitable to handle complex data types (including arithmetic calculus for example). In our case studies, when the requirements include numerics, we use a simple kind of abstract interpretation [14], abstracting a number by a set of intervals.

Restriction on postconditions. In our model, we have restricted the usage of the postconditions: The postconditions must be deterministic. Though this restriction is a limitation of our model, conditional postconditions can still be expressed, making the condition explicit. Let us illustrate that with the example of a use case $\mathrm{A(x : X)}$ which results in the predicate $\mathrm{p1}$ or the predicate $\mathrm{p2}$, depending on a given condition $\mathrm{c(x)}$. The postcondition "$\mathrm{p1}$ or $\mathrm{p2}$" is not acceptable from our point of view: It is not deterministic, since the condition $\mathrm{c(x)}$ does not appear in the postcondition. However, the postcondition can be expressed as follows: "$\mathrm{c(x)@pre}$ implies $\mathrm{p1}$ and not $\mathrm{c(x)@pre}$ implies $\mathrm{p2}$." In this latter postcondition, the condition $\mathrm{c(x)}$ is made explicit, and the postcondition is thus valid.

3.2 Simulating the Use Cases

Using the contracts, the use cases can be simulated: That means that we are able to decide which use cases with which parameters can be applied from a given simulation state. In fact, the idea is to "instantiate" the use cases with a set of values (the actual parameters, i.e., the actual actors and business concepts) replacing their formal parameters. In the virtual meeting example, we want to obtain the ordering of the use cases with two meetings $m1$ and $m2$, and two participants $p1$ and $p2$. The instantiated use cases of $open(u : participant; m : meeting)$ are $open(p2, m1)$, $open(p1, m2)$, and $open(p2, m2)$. In the following, we call *instantiated use cases* (respectively, *predicates*) the set of use cases, (respectively, predicates) obtained by replacing their sets of formal parameters with all the possible combinations of their possible actual values.

We define a simulation state as a set of instantiated predicates: these which are valuated to *true*. For example, the state $\{connected(p1), connected(p2)\}$ corresponds to a state of the simulation for which participants $p1$ and $p2$ are connected.

Applying an instantiated use case. An instantiated use case iuc can be applied from a simulation state s when s logically implies the precondition of iuc. Then, if iuc is applied, the current state is modified according to the postcondition of iuc: s is modified such as the postcondition of iuc: s is implied by the new current state. For example, if we apply the instantiated use case $close(p1, m1)$ (cf. Fig. 2) from the state

$$s = \{connected(p1), manager(p1, m1), moderator(p1, m1), created(m1), opened(m1), entered(p1)\},$$

the new current state will be

$$s' = \{connected(p1), manager(p1, m1), moderator(p1, m1), created(m1), closed(m1)\}.$$

To be able to compute the new current state, we use the restriction on the postcondition explained in the previous section: Since the postconditions must be deterministic, applying a postcondition will deterministically lead to a single new state.

3.2.1 Construction of the Simulation Model

To simulate a use case system, we need an initial state and the enumeration of all the business entities present in the system, in order to be able to compute the instantiated use cases and predicates. As an example, to deal with two participants and two meetings, we can declare

$$\mathrm{p1, p2 : Participant}$$
$$\mathrm{m1, m2 : Meeting.}$$

3.2.2 Benefits and Limitations of the Simulation

A simulation tool allows the requirement analyst to check her requirements: The simulator is an interactive tool proposing the user a list of all the instantiated use cases that can be applied from the current simulation state (starting from the initial state). It simulates the execution of a selected instantiated use case, thus producing a new current state. Using such a tool, the requirement analyst can check whether the requirements she has specified conform with the ones she had in mind, comparing the obtained behavior with the expected behavior. Inconsistencies between predicates and contracts can be identified, as well as underspecification or errors in the requirements. Using our simulation tool, properties can also be verified using model-checking techniques. For example, invariants can be checked: In the virtual meeting system, one can check that it is not possible that several participants speak at the same time in a meeting. It is expressed using the following invariant:

$$\mathrm{not\ exists(u1, u2 : participant; m : Meeting)}$$
$$\mathrm{\{\ u1 /= u2\ and\ speaker(u1, m)\ and\ speaker(u2, m)\ \}.}$$

Moreover, the tool can exhibit a path leading to a given configuration.

Since there is no predicate calculus in this simulation tool, its main limitation relates to predicate dependencies. When a use case U has "*a and b*" as precondition, while the predicate c is true in the current state, if it happens that $c = (a\ and\ b)$, then our simulator would not propose to apply U. This is because we do not provide ways to define relations between predicates ($c = (a\ and\ b)$).

3.3 Exhaustive Simulation and Transition System

Defining contracts for each use case allows ordering dependencies among use cases to be inferred. In our model, if there is no explicit dependency between two use cases, then these use cases can be executed in parallel. A representation of the valid sequences of the use cases is built by exhaustively simulating the system. It results in a transition system called Use Case Transition System (UCTS). Formally, the UCTS is defined by a quadruple $(Q, q_0, A, \hookrightarrow)$, where

- Q is a finite and nonempty set of states, each state being defined as a set of instantiated predicates,
- q_0 is the initial state,
- A is the alphabet of actions, an action being an instantiated use case, and
- $\hookrightarrow \subseteq Q \times A \times Q$ is the transition function.

The states of the UCTS represent an abstraction of the states of the system: A state of the UCTS is a reachable combination of values of predicates. Each transition, labeled with an instantiated use case, represents the execution of an

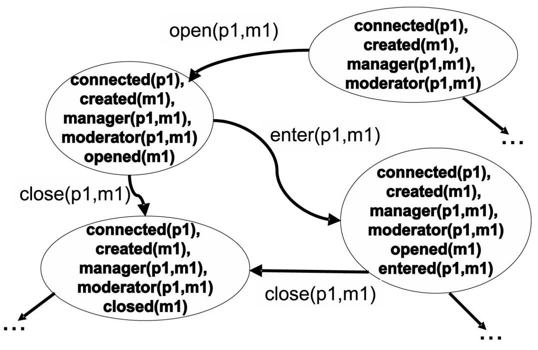

open(p1,m1)
connected(p1), created(m1), manager(p1,m1), moderator(p1,m1)
connected(p1), created(m1), manager(p1,m1), moderator(p1,m1) opened(m1)
enter(p1,m1)
close(p1,m1)
connected(p1), created(m1), manager(p1,m1), moderator(p1,m1) opened(m1) entered(p1,m1)
connected(p1), created(m1), manager(p1,m1), moderator(p1,m1) closed(m1)
close(p1,m1)

Fig. 3. Extract of the UCTS for the virtual meeting.

instantiated use case. A path in the UCTS is, thus, a valid sequence of use cases. A partial UCTS obtained for the virtual meeting example is given in Fig. 3.

Such a UCTS is built from the simulation model using Algorithm 1, which successively tries to apply each instantiated use case from the current state (initially, from the initial state). Applying a use case is possible when its precondition is implied by the set of true predicates contained in the label of the current state and leads to creation of an edge from the current state to the state representing the system after the postcondition is applied. The algorithm stops when all the reachable states have been explored.

Algorithm 1 (Algorithm Producing the UCTS)

```
algorithm buildUCTS
param initState : STATE
        useCases : SET[INST_UC]
var
    result: UCTS
    to_visit : STACK[STATE]
    currentState : STATE
    newState : STATE
init
    result.initialState $\leftarrow$ initState
    to_visit.push(initState)
body
    while (to visit $\neq \emptyset$ do
      currentState $\leftarrow$ to_visit.pop
      $\forall$ uc $\in$ useCases | currentState$\Rightarrow$ uc.pre do
        newState $\leftarrow$ apply(currentState, uc)
          if newState $\notin$ result then
            result.Q $\leftarrow$ result.Q $\cup$ {newState}
            to_vist.push(newState)fi
            result.$\hookrightarrow$ $\leftarrow$ result.$\hookrightarrow$ $\cup$
                {(currentState,uc,newState)}
        done
    done
end
return result

function apply(currentS: STATE, uc: USECASE):STATE
// returns the new current state obtained when the
// instantiated use case uc is applied from state currentS
```

3.3.1 Limitations of the UCTS Model

Size of the UCTS. The theoretical complexity of the UCTS, both in building time and in memory, may be quite high. In practice, however, if the engineers follow widely accepted methodological guidelines (cf. [15]), for most systems the number of high level use cases is smaller than 20, so the UCTS size remains reasonable (see [16]). In fact, the theoretical maximal size $maxsize_{UCTS}$ of a UCTS depends on the number n_{ip} of instantiated predicates: $maxsize_{UCTS} = 2^{n_{ip}}$ with $n_{ip} = p \times (max_{instances})^{max_{param}}$, where p is the number of predicates, $max_{instances}$ is the maximum number of instances, and max_{param} is the maximum number of parameters per predicate. However, in practice, many of the potential states are not reachable. For example, for the virtual meeting system with two meetings and two participants, $maxsize_{UCTS} = 2,097,152$, but the actual size of the UCTS is 1,616, and for the airborne systems we studied, the size of the UCTS is also in the range of 10^3. The real complexity of the UCTS also depends on the complexity of the contracts: disjunctions in the preconditions increase the number of transitions between the states, and disjunctions in the postconditions increase the number of reachable states.

For the systems we studied, even if a large number of instances might be present in the real system, only a small number of instances are necessary for achieving statement coverage and, thus, the size of the UCTS remains acceptable. This is due to the fact we focus on pure functional testing and not, for example, load testing or performance testing: for this kind of testing, many instances are needed and our UCTS model is not adapted (this problem is a classical one when dealing with enumerated models and not symbolic models). Also, the number of instances sufficient for a statement coverage purpose is not always easy to determine, while the efficiency of the generated tests depends on this number of instances.

Managing the concurrency. The UCTS does not allow modeling of true concurrency (contrary to other models, such as by Jard [17]): We use an interleaving semantics to represent the potential parallelism between use cases. That means that when two use cases A and B can be applied in parallel, in the UCTS it is translated by two paths: A followed by B, or B followed by A.

3.4 Test Objective Generation with Regard to a Coverage Criterion

A UCTS is a representation of all the possible orderings of the use cases. From a UCTS, the aim is to generate test objectives with regard to a given UCTS coverage criterion. Four functional **structural criteria** are defined in this section to cover a UCTS, as well as one **functional semantic criterion**, and one **robustness semantic criterion**.

Definition 1. A test objective *is defined as a finite sequence of instantiated use cases. In most cases, a test objective is not a test case in the sense that it cannot be directly executed on an implementation: A test case generator (for example, a test synthesis tool [18], [19]) has to be used to go from test objectives to test cases.*

An example of test objective for the virtual meeting example is:

$$[open(p1, m1), enter(p2, m1), leave(p2, m1), close(p1, m1)].$$

Definition 2. *A set of test objectives is said to be consistent with a UCTS iff each test objective exercises a path of the UCTS. A* *path in the UCTS is here defined as the classical notion of path in a graph, the first vertex corresponding to the initial state.*

All Edges criterion (AE). A set of test objectives *TOs* satisfies the *all edges* coverage criterion for a use case transition system *ucts* iff each edge involved in *ucts* is exercised by at least one test objective from *TOs,* more formally iff

$$\forall t \in \hookrightarrow, \exists to_i \in TOs, t.action \in to_i,$$

where $t.action$ corresponds to the label of transition t, i.e., an instantiated use case.

All Vertices criterion (AV). A set of test objectives *TOs* satisfies the *all vertices* coverage criterion for a use case transition system *ucts* iff each vertex v involved in *ucts* is exercised by at least one test objective from *TOs* (i.e., when a transition leading to or incoming from v is exercised), more formally iff

$$\forall q \in Q, \exists to_i \in TOs, \exists iuc \in to_i, \exists t \in \hookrightarrow,$$
$$t.action = iuc \wedge (t.origin = q \vee t.dest = q),$$

where $t.origin$ (respectively $t.dest$) corresponds to the state from which t is outgoing (respectively to which t is incoming), and *iuc* is an instantiated use case.

All Instantiated Use Cases criterion (AIUC). A set of test objectives *TOs* satisfies the *all instantiated use cases* coverage criterion for a use case transition system *ucts* iff each instantiated use case of the system is exercised by at least one test objective from *TOs*, more formally iff

$$\forall iuc \in IUC, \exists to_i \in TOs, \exists iuc_{to} \in to_i, \exists t \in \hookrightarrow,$$
$$iuc = iuc_{to} \wedge iuc_{to} = t.action,$$

IUC being the set of all the instantiated use cases of the system.

The AE criterion guarantees that all the possible actions from all the possible states of the requirements are exercised at least once. The AV criterion guarantees that all the states are reached during the execution of the test objectives. Since it does not even guarantee that all the use cases are exercised once, we combined it with the AIUC criterion, which ensures that all the possible use cases are exercised. The AV-AIUC criterion is, thus, a combination of AV and AIUC.

All Vertices and All Instantiated Use Cases criterion (AV-AIUC). A set of test objectives *TOs* satisfies the *all instantiated use cases* coverage criterion for a use case transition system *ucts* iff each instantiated use case of the system and each vertex involved in *ucts* are exercised by at least one test objective from *TOs*, more formally iff *TOs* satisfies AIUC and AV.

All Precondition Terms criterion (APT). A set of test objectives *TOs* satisfies the *All Precondition terms* criterion for a contracts system iff each use case is exercised in as many different ways as there are predicates combinations to make its precondition true. More formally, for each $uc \in set_{uc}$ where set_{uc} is the set of use cases of the system, let E be the set of valuations making $uc.pre$ true independently from the actual parameters of the predicates. Let set_{iuc} be the set of instantiated use cases of the system. Then, *TOs* satisfies the *All Precondition terms* criterion iff

$$\forall uc \in set_{uc} \forall (e \in E \mid \exists q \in Q, q.label \Rightarrow e),$$
$$\exists t \in \hookrightarrow, t.action \in uc.set_{iuc} \wedge t.origin.label \Rightarrow e,$$

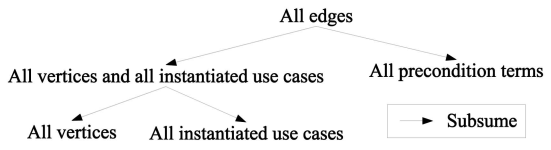

All edges
All vertices and all instantiated use cases
All precondition terms
All vertices
All instantiated use cases
Subsume

Fig. 4. Subsume relations among the five functional criteria.

where $q.label$ is the label attached to the state q, i.e., a set of predicates, and e is a set of true instantiated predicates.

The idea of the *All Precondition Terms* criterion is to guarantee that all the possible ways to apply a use case are exercised. A use case can be applied iff its precondition is true; this precondition being a logical expression on predicates, there are several valuations for the predicates which make it true (as an example, if a precondition is $a \; or \; b$, three valuations for the couple (a, b) must be exhibited $((true, true), (true, false), (false, true))$. The criterion *All Precondition Terms* will find sequences of use cases such that each use case is applied with all the reachable valuations of the expression $(precondition = true)$. Our algorithm to satisfy the APT criterion first computes the set E of valuations for this expression, independently from the parameters. Then, states are found in the UCTS for which such valuations hold. Finally, a path leading to such states and applying the studied use case is extracted.

The subsume relations among the five criteria are given in Fig. 4. (A criterion C_1 is said to subsume another criterion C_2 iff, for each use case system and for each set S of test objectives satisfying C_1, S also satisfies C_2.)

The two criteria AE and AV are classical graph coverage criteria, and AIUC aims at covering all the labels of the labelled transition system. However, the AE criterion leads to generating far too many test objectives,[1] and criteria AV and AIUC are too weak (see Section 5). We have, thus, created the AV-AIUC criterion. The APT was chosen because it is a different way (more semantic than structural) of defining coverage, and it requires a small number of test objectives (see Section 5).

The test objectives generated with the criteria presented above are correct sequences of use cases. They are expected to ensure that the requirements expressed by the use cases are not violated by the system under test but do not ensure that the system is able to detect that one of its requirements is violated. In other words, these criteria do not provide robustness test objectives. However, as soon as the requirements are precise enough, the generated UCTS can be used as an oracle for robustness tests. The principle is to generate valid paths that lead to an invalid application of a use case. The idea is to correctly exercise the system up to a given point, and then to apply a nonspecified action. The execution of such a robustness test must be detected (and rejected) by the system. If not, a robustness weakness has been detected. The goal is thus to test the robustness/defensive code of the system as much as possible at this stage. The difficulty is to define an adequate criterion. The criterion we use to generate robustness paths with the UCTS is close to the *All Precondition Terms* one: For each use case, it selects all the shortest paths leading to each of the possible valuations that violate its precondition.

1. As an example, for the virtual meeting system (presented in Section 5), which is made of 2,500 LOC, the AE criterion leads to 13,841 test objectives.

Robustness criterion: A set of test objectives *TOs* satisfies the robustness criterion for a contracts system iff each use case is exercised in as many different ways as there are predicate combinations to make its precondition false. More formally, for each $uc \in set_{uc}$ where set_{uc} is the set of use cases of the system, let E be the set of valuations making $uc.pre$ false. Let set_{iuc} be the set of instantiated use cases of the system. Then,

$$\forall uc \in set_{uc} \forall (e \in E | \exists q \in Q, q.label \Rightarrow e),$$
$$\exists t \in \hookrightarrow, t.action \in uc.set_{iuc} \wedge t.origin.label \Rightarrow e,$$

where $q.label$ is the label attached to the state q, i.e., a set of predicates.

The robustness test objectives will test the defensive code of the application, which is not tested with the functional tests previously generated. Joining the two sets of tests, we can test that the application does what it should (according to the requirements) and also that it does not do what it should not (which is very important in the case of airborne weapons systems). The criteria are compared and discussed on three case studies in Section 5.

Algorithms (implemented in our prototype tool) exist for each of the proposed criteria that produce a set of consistent test objectives satisfying the criterion. As an example, the algorithm to produce a set of test objectives consistent with the *all precondition terms* criterion is given in Algorithm 2. All these algorithms are based on a breadth-first search in the UCTS, from its initial state. Such a technique ensures that the obtained sets of test objectives are consistent with the considered UCTS. The result of our algorithms depends on the order in which the nodes are visited; they thus contain some indeterminism, even if our implementation is totally deterministic. The choice of a breadth-first search is made in order to obtain small test objectives: Small tests are more meaningful and understandable by humans than large ones. This is especially important at requirement stage, which should remain a high-level view. Using a breadth-first search algorithm ensures that the size of the computed paths is minimal, but does not ensure that the number of paths found is minimal.

Algorithm 2 (Algorithm producing the set of test objectives satisfying the all precondition terms criterion)

```
algorithm buildTestObjectives_with_APT_criterion
param
  ucts : UCTS, set_uc : LIST[INST_UC]
body
  ∀ uc ∈ set_uc do
    var set_b : LIST[BoolExpr]
  set_b ← getAllTrueValuations(uc.pre)
  ∀ e ∈ set_b do
    if (getPath(e,ucts).dumpIUC( )) ≠ ∅ then
        result.add(getPath(e,ucts).dumpIUC) fi
      done
  done
return result
end

function getAllTrueValuations
param b:BoolExpr
return LIST[BoolExpr]
// returns all the valuations making b true, in the form of
// Boolean expressions

function getPath
param b:BoolExpr, ucts:UCTS
return list[EDGES] ∪ ∅
// returns the first ucts path found leading to a state where
// b is true, or an empty list if such a path cannot be found

function
  function  Path::dumpiuc:LIST[INST_UC]
// returns the list of instantiated use cases present
// in the path.
```

4 Generating Test Cases from Test Objectives and Sequence Diagrams

The test objectives are generated using the use cases. They have then to be transformed into valid sequences of calls and expected outputs on the system under test. In this section we detail the second part of the method, which aims at generating test scenarios from test objectives and use case scenarios.

As already explained, we define a *test scenario* as a sequence diagram representing a test. Test scenarios may differ from the test cases in the fact that the test cases can be applied directly with a test driver, whereas the test scenarios may still be incomplete. The test scenarios contain the main messages exchanged between the tester and the system under test.

We propose to derive test scenarios from test objectives using the use case scenarios. Each use case is documented by its contracts and scenarios illustrating how the system has to be stimulated by the actors in order to perform the use case and how the system should react to the stimulation. We assume that these scenarios are expressed with UML sequence diagrams.

4.1 Motivation of Using Sequence Diagrams

Our first motivation to use scenarios is to improve the oracle. The test objectives built with the contracts method do not embed a precise oracle. The oracle embedded is just the expectation of a noninterrupted execution for the functional test objectives, either of an error or of a warning for the robustness test objectives.

Such verdicts are limited since they check neither the system output's consistency nor any property of the system state. This kind of information cannot be found in the use case contracts, since they are high level and independent of the rest of the modeling—in the sense that they do not refer to any other modeling element. But this information can typically be found in the sequence diagrams illustrating each use case. Each of these sequence diagrams illustrates how an actor stimulates the system, and how the system responds, thus sequence diagrams can be used to refine the oracles. As an example, for the virtual meeting system, it is difficult to express at the use case level the fact that it is not possible to plan a meeting for a certain date when one of its expected participants is not available at this date (i.e., if their calendar already has an entry at this date). Indeed, the contracts do not capture complex objects such as the calendars and the dates. Nevertheless, it is easy and natural to express it with a use case scenario relying on the rest of the model.

Our second motivation is to obtain test scenarios from which a code generator can generate the test cases. The test objectives generated are very far from the messages exchanged during the test, since they just consist of sequences of parameterized use cases. The communication protocols are unknown at this stage. Using the sequence diagrams attached to the use cases allows us to bridge part of the gap between the test objectives and the test cases, since the use case scenarios describe the expected exchanges of messages between the actor and the system. In particular, they contain information on the types of the involved objects. While the test objectives generated using to the use cases involve types that represent actors, or business concepts, the sequence diagrams contain references to types existing in the UML analysis model of the system (the analysis class diagram).

Our third motivation comes from the fact that scenarios and, in particular, sequence diagrams are increasingly being used in industry in the early phases of requirements. The conclusion of the survey of industrial software projects [20] insists on the industrial need to base system tests on use cases and scenarios, and explains that most projects lack a systematic approach to define test cases based on scenarios. Our approach is a proposal to make an easier use of scenarios in the validation phases.

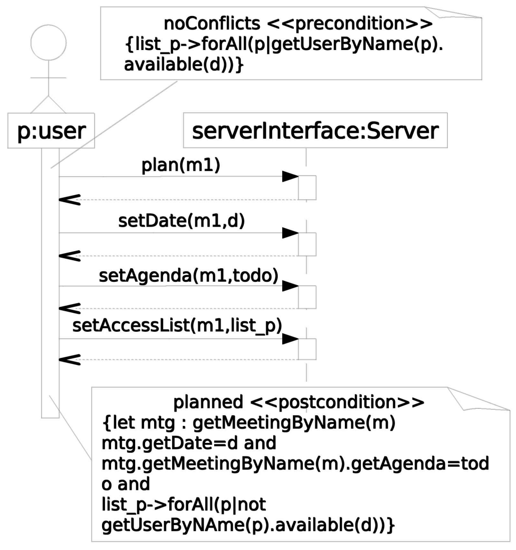
4.2 From Test Objectives to Test Scenarios: Method

We propose to replace the instantiated use cases with instantiated use case scenarios in the test objectives. Sequences of scenarios are thus obtained and are transformed into test scenarios using *strong sequential composition* (the strong sequential composition implies that every event of a use case scenario is executed before an event of the next scenario can be executed [21]). In the following, we present the sequence diagrams we deal with, the test structure generated, and the way it is generated.

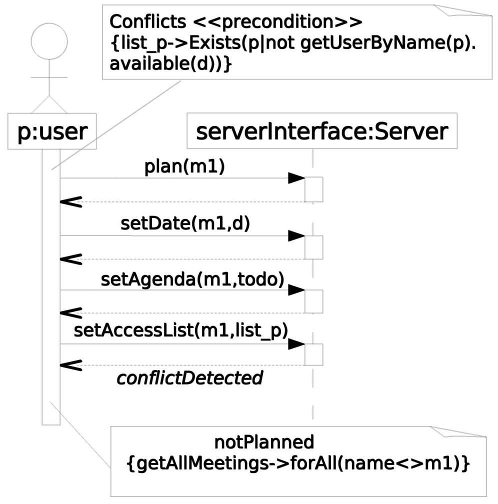
4.2.1 *Sequence Diagrams*

Each of the sequence diagrams we deal with is attached to a use case and represents one of its nominal or exceptional scenarios. The sequence diagrams are system-level in the sense that they only involve the system itself and the actors.

These sequence diagrams may involve parameters: Since they are attached to parameterized use cases, it is quite natural to find in the sequence diagrams at least the same parameters as in its owner use case. In the following, we assume that a use case scenario uses exactly the same parameters as its use case. The sequence diagrams contain more information than the use case, since they rely on other model elements, and they are more precise than the use case they describe. The use case scenarios can thus contain more detailed pre and postconditions than the ones of the owner use case. Those contracts are of two types: Those which rely on the rest of the model (they are written in OCL [22], [23]), and those which rely on the predicates introduced in the use case analysis (they are written in the form of a logical expression with the use case contracts language and are especially useful for the exceptional use case scenarios: they define the state in which an exceptional behavior may occur).


noConflicts <<precondition>>
{list_p->forAll(p|getUserByName(p).
available(d))}
p:user
serverInterface:Server
plan(m1)
setDate(m1,d)
setAgenda(m1,todo)
setAccessList(m1,list_p)
planned <<postcondition>>
{let mtg : getMeetingByName(m)
mtg.getDate=d and
mtg.getMeetingByName(m).getAgenda=tod
o and
list_p->forAll(p|not
getUserByNAme(p).available(d))}

(a)



(b)

Fig. 5. Examples of scenarios the use case *plan*. (a) Nominal. (b) Exceptional.

Nominal use case scenarios represent the basic ways to successfully exercise a use case. Exceptional use case scenarios represent ways to exercise a use case leading to a failure, the raise of an exception, or an error message: exceptional use case scenarios make the use case fail. In our context, the nominal use case scenarios, owning the tagged value *{nominal}*, will be used for functional testing and the exceptional ones, owning the tagged value *{exceptional}*, for robustness testing.

To sum up, the sequence diagrams we deal with are system-level; they may involve parameters and own additional pre and postconditions of two types: relying on the rest of the model (written in OCL) and relying on the use case predicates.

Fig. 5 provides a nominal and an exceptional use case scenario for the use case *plan* of the Virtual Meeting system. In the two scenarios, *d* and *list_p* are parameters of the use case *plan*, which, respectively, designate the date and the list

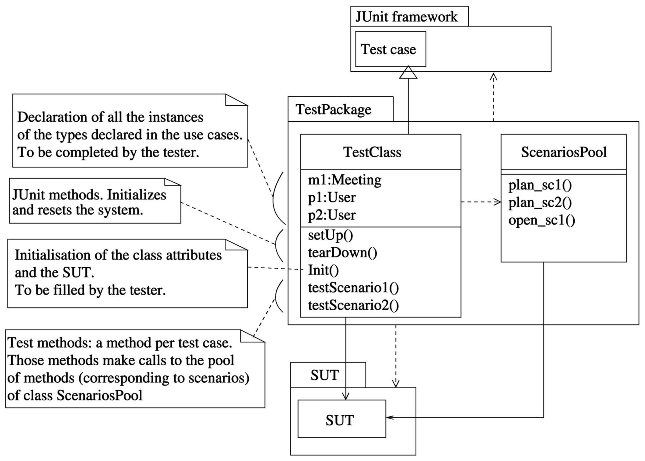

JUnit framework
Test case
TestPackage
TestClass
m1:Meeting
p1:User
p2:User
setUp()
tearDown()
Init()
testScenario1()
testScenario2()
ScenariosPool
plan_sc1()
plan_sc2()
open_sc1()
SUT
SUT
Declaration of all the instances of the types declared in the use cases. To be completed by the tester.
JUnit methods. Initializes and resets the system.
Initialisation of the class attributes and the SUT. To be filled by the tester.
Test methods: a method per test case. Those methods make calls to the pool of methods (corresponding to scenarios) of class ScenariosPool

Fig. 6. Structure of the generated test material.

of the invited participants of the meeting being planned. The nominal precondition is an OCL precondition that asserts whether the invited participants are available at the meeting date. The nominal postcondition checks that the meeting has been planned with the correct parameters. The exceptional use case scenario checks that the participants are not available at the meeting date in its precondition, and that the meeting is not planned in its postcondition.

4.2.2 Structure of the Generated Test Material

Our prototype tool *UC-SCSystem* generates test cases in the form of Java classes, using the JUnit framework [24] as a test harness: it runs all the tests, and it provides the test launcher and facilities to write assertions on the system. A class diagram of the generated test structure is given in Fig. 6.

4.2.3 Building a Pool of Use Case Scenarios

The class *ScenarioPool* contains all the use case scenarios: For each of the use case scenarios, a static method is created in *ScenarioPool*. The formal parameters of this method are the parameters of the scenario, i.e., the parameters of the corresponding use case. Each such methods is documented by the test scenario itself. The test scenario is transformed into Java code: The OCL contracts are transformed into JUnit assertions, and the exchanges of messages into Java calls (note that we did not concentrate on this part of the problem, we just rely on existing approaches [25], [26]). As an example, the code for the nominal *plan* scenario is given in Fig. 7.

4.2.4 Building the Test Scenarios

The attributes of the main test class are the instances the tester wants to deal with. The main test class also owns a set of methods with names beginning with "test": each corresponds to a test case. It is documented with a test scenario, i.e., a sequence diagram, which is built as follows:

Building a test scenario from a test objective consists of replacing the use cases with their corresponding scenarios, and in composing the scenarios using strong sequential composition.

In concrete terms, the code of the methods representing a test case is composed of successive calls to the methods

```
public static void planNominal(String Meetingname,Date date, String agenda,
                        String[] access_list){
  try {
     for (i=0,i<access_list.size,i++){
     assertTrue((User)system.getUserByName(access_list[i]).available(date))}
    system.execute(plan(MeetingName));
    system.execute(setDate(MeetingName, d));
    system.execute(setAgenda(MeetingName, agenda);
    system.execute(setAccessList(access_list));
     for (i=0,i<access_list.size,i++){
      assertFalse((User)system.getUserByName(access_list[i]).available(date))};
  } catch (Exception e){e.printStackTrace();}
}
```

Fig. 7. Java code for the nominal scenario of the use case *plan*.

corresponding to the sequence diagrams in the *scenariosPool* class, with the correct actual parameters.

Definition 3. *An* instantiated scenario *is a scenario whose formal parameters are replaced by actual parameters.*

When an instantiated use case is replaced by a use case scenario, this scenario is instantiated using the actual parameters of the instantiated use case.

Let $\{scn_{i,j}\}_{,j\in 1,..n}$ be the set of n nominal scenarios attached to the use case uc_i and let $\{sce_{i,j}\}_{,j\in 1,..m}$ be the set of m exceptional scenarios attached to the use case uc_i. We denote $\circ$ the sequential composition of scenarios. We denote $\times$ the Cartesian product; the Cartesian product of two sets A and B is defined as: $\{(a,b)\,|\, a \in A \wedge b \in B\}$.

A test scenario is defined from a tuple of use case scenarios $(sc_1, ..., sc_n)$ as $sc1 \circ ... \circ sc_n$ (the strong sequential composition of the elements of the tuple). The set of tuples defining a set of test scenarios $TS = \{ts_1,\ ...,\ ts_u\}$ obtained from a test objective to is denoted TS_{tuple}. They are obtained using Cartesian products on sets of instantiated scenarios, as explained below. The use case scenarios are instantiated using the *inst* method, taking as parameter an instantiated use case.

Functional test scenarios. A test objective $to = [uci_1..uci_t]$ is transformed into the set of tuples TS_{tuple} defined as:

$$\begin{aligned} TS_{tuple} &= \prod_{i=1}^{t} \{scn_{i,j}.inst(uci_i)\}_{j\in 1,\dots,n} \\ &= \{scn_{1,j}.inst(uci_1)\}_{j\in 1,\dots,n} \times \dots \\ &\quad \times \{scn_{t,j}.inst(uci_t)\}_{j\in 1,\dots,n}. \end{aligned}$$

Building the functional test scenarios can be seen as replacing each of the instantiated use cases of to by each of its nominal scenarios. Once all the instantiated use cases have been replaced, a tuple of scenarios is obtained, and strong sequential composition is achieved to obtain a test scenario.

Robustness test scenarios. A test objective $to = [uci_1..uci_t]$ is transformed into the set of tuples TS_{tuple} defined as

$$\begin{aligned} TS_{tuple} &= \prod_{i=1}^{t-1} \{scn_{i,j}.inst(uci_i)\}_{j\in 1,\dots,n} \times \{sce_{t,j}\}_{j\in 1,\dots,m} \\ &= \{scn_{1,j}.inst(uci_1)\}_{j\in 1,\dots,n} \times \\ &\quad \dots \times \{scn_{t-1,j}.inst(uci_{t-1})\}_{j\in 1,\dots,n} \\ &\quad \times \{sce_{t,j}.inst(uci_t)\}_{j\in 1,\dots,m}. \end{aligned}$$

Building the robustness test scenarios can be seen as replacing the instantiated use cases of to by its scenarios.

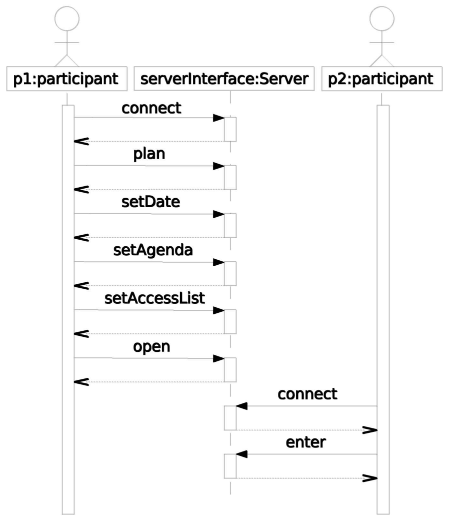


Fig. 8. An example of a generated test case.

The process to replace the $t - 1$ first instantiated use cases is the same as for functional test scenarios, and the last instantiated use case is replaced by its exceptional scenarios.

In Fig. 8, an example of a functional test scenario is given: A participant connects to the server, plans a meeting, and opens it. Then, another participant enters the meeting.

For both exceptional and functional treatments, when a use case is replaced by one of its scenarios, the preconditions based on predicates of this scenario are used to check whether the use case is really replaceable by this scenario. A use case scenario can replace a use case at a certain stage of execution iff the state reached at this stage logically implies the precondition of the use case scenario. If a use case scenario cannot replace its use case, it is discarded for this test objective. It may even happen that no use case scenario can replace a use case in a test objective; in this case, the test objective is discarded, and the test generation tool reports it.

4.3 Verdict Analysis

We generate test scenarios using the use case scenarios. Since these scenarios are requirements on the behavior of the system under test, it is natural to expect that all the use case scenarios are exercised by at least once by the tests. We thus propose the following intuitive test strategy.

All scenarios strategy. Let *TCs* be a set of test cases, the all_scenarios test strategy consists of ensuring that each use case scenario expressed in the requirements is executed correctly at least once by test cases in *TCs*. A use case scenario is said to be executed correctly by a test case when the part of the test case corresponding to the scenario is executed without detecting an error.

Successfully applying this strategy means that all the behaviors specified in the requirements have been exhibited in the system by at least one execution of a test case. Determining if the *all scenarios strategy* has been successfully applied can only be done after the execution of all the test cases and depends on the verdicts they have produced.

4.3.1 The Verdicts

The test cases contain an oracle that allows them to emit three types of verdicts: pass, fail, or inconclusive. They are

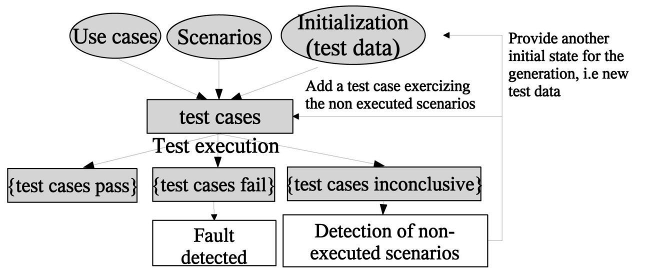

Use cases
Scenarios
Initialization (test data)
Provide another initial state for the generation, i.e new test data
Add a test case exercizing the non executed scenarios
test cases
Test execution
{test cases pass}
{test cases fail}
{test cases inconclusive}
Fault detected
Detection of non-executed scenarios

Fig. 9. Verdict analysis.

determined by executable assertions derived from the OCL contracts.

Fail verdict. The fail verdict is emitted when a postcondition is violated during execution. The postconditions check that the system reacted correctly ensuring simple properties from the interface of the system. If the system did not react correctly, the test case fails: An error has been detected, and the test report makes it clear where the test case failed.

Pass verdict. The pass verdict is emitted when the test case is correctly and totally executed; it indicates successful test cases.

Inconclusive verdict. The inconclusive verdict is emitted when a test case is aborted due to a precondition evaluated to false. It thus detects nonexecuted test scenarios. The preconditions add requirements on the execution of the sequence diagram and, thus, are guards on its continuation. During execution, if such a guard is evaluated to false, the test case must stop, but it does not necessarily mean that an error has been detected. It might mean that the instantiated sequence chosen to implement an instantiated use case is incorrect: This latter case may come from either underspecification (or from errors in the specification), or from the test data used in the instantiated use case scenario. In order not to reject a correct implementation, we then introduce an *inconclusive* verdict and notify the tester that a particular test scenario was not executed. It corresponds to a failure of the test strategy, since one scenario could not be executed successfully. When it is possible, the tester should create new instantiated use case scenarios with values allowing the test objective to be executed. It may be guided by specific tools as test synthesis tools [18], [19] and constraint solver [27].

4.3.2 Reaching the All Scenarios Strategy

As illustrated in Fig. 9, after the execution of the test cases, one has to determine whether the *all scenarios strategy* is successfully applied. If all the use case scenarios describing the system have been exercised without emitting inconclusive verdicts, then the all scenarios test strategy is successful. It means that the system is consistent with its specification in the sense that all the use case scenarios describing the system have been exercised. In the other case, that means that the set of test cases has to be improved. Two solutions can be used to reduce the number of inconclusive verdicts. First, a pragmatic solution consists of manually creating new test cases exercising the remaining nonexercised use case scenarios. Second, the new test cases can be generated using another initial state, i.e., modifying the test data. The new initialization has to be written studying the preconditions that failed, in order to

TABLE 1
Code Repartition for Our Case Studies

	ATM	FTP	Virtual Meeting
Dead code	0%	3.7%	9%
Robustness w.r.t env	0%	5.8%	8%
Robustness w.r.t. spec	9.09%	17.8%	18%
Functional	90.91%	72.7%	65%

deduce which test data would have satisfied them. This phase is manual and left to the tester. It could be automated using a constraints solver [27] that can find the test data satisfying the preconditions.

5 Results and Discussion

This section presents an empirical evaluation of the approach, based on the results obtained with three small case studies. We first present the systems under test, and then we briefly describe the experimental protocol and give statistics on the generated test cases. Finally, we compare the criteria and study the efficiency of the generated test cases in terms of statement coverage. While the number of covered statements is a very crude measure of test efficiency, it is widely used in industrial circles because it is easy to measure even for real-time embedded OO software, where a strict mapping is mandatory from requirements to code. On the other hand, there is, for instance, no simple mapping between requirements and data flows, especially in an OO system.

5.1 The Case Studies

To perform an analysis of the category of code one can expect to cover, we extensively analyzed three programs (between 800 LOC and 2,000 LOC). Our objective here is to determine whether our approach is efficient to cover the main functional code. To do that, we classify the code into the following categories:

- Dead code: Some of our case studies have dead code, which for instance consists of relevant but yet unused accessors. Functional testing cannot deal with this code: it has to be tested during the unit testing. For the following studies, we removed this dead code to focus on the efficiency of our tests on reachable code.
- Robustness code with regard to the specification which asserts that incorrect or invalid demands are detected and rejected (for example, code impeding a nonauthorized user's performing a given action).
- Robustness code with regard to the environment, which asserts that the inputs coming from the environment are correct (for example, code handling input/output or network exceptions).
- Functional code.

The case studies are the following:

1. An Automated Teller Machine (ATM). We adapted the application taken from Bjork [28] by decoupling the core from the graphical interface. It provides the following main functions: consultation, withdrawal, deposit of checks and cash, and transfer from account to account. The implementation we use is composed of 850 lines of code (and 186 executable statements.)[2]
2. An FTP server taken from [29]. We studied a large part of this server (the most interesting commands), the implementation is composed of 500 lines of code (and 207 executable statements).
3. The server of virtual meetings (VM), used as an ongoing example in this paper. Let us recall that it allows virtual meetings to be organized, with a manager and a moderator. The participants can enter the meetings and speak into them, under certain conditions depending on the meeting type. The system contains 2500 lines of code (and 780 executable statements). This system is used for a software engineering course in the University of Rennes 1.

TABLE 2
Statistics of the Generated Test Cases

	ATM	FTP	VM
# use cases	5	14	14
# nominal UC-scenarios	5	14	14
# exceptional UC-scenarios	17	14	14
# generated functional test cases	6	14	15
# generated robustness test cases	17	33	65

For each case study, the repartition of code into the four categories is given in Table 1.

While the ATM has neither code for dealing with implementation dependent misuses (robustness with regard to environment) nor dead code, the FTP and VM case studies have between 9 percent and 17 percent of this kind of code: It cannot be covered by test cases generated with our approach. More generally, it means that a model-based testing approach must be completed by a unit testing stage for this environment-dependent code. Concerning the proportion of code dedicated to "robustness" with regard to specification, it varies from 9 percent (for the ATM) to 18 percent. The remaining code concerns the code implementing the functions described by the use cases.

For each case study, the experimental protocol we applied consists of

- a requirement stage: Being given the set of use cases, we define and attach scenarios to each use case (at least one exceptional and one nominal scenario per use case). Then, contracts are expressed for each use case and we use a simulation process to check if the set of use case contracts is consistent and contains enough information.
- a test generation stage: Based on each of the proposed coverage criteria, test objectives are generated. By the same way test cases are produced by replacing use cases by scenarios in the test objectives. Table 2 presents the number of use cases, use case scenarios and the corresponding numbers of test objectives and test cases.
- a test execution stage: A test driver has been developed to launch the test cases (into the JUnit

2. The count of executable statements does not include declarations (such as class declarations, attribute declarations, operation declarations, or input statements).

TABLE 3
Statistics on Test Objectives (TOs) Generated for the Three Case Studies with Our Five Criteria

	Virtual Meeting		ATM		FTP	
Criterion	#TOs	av. size	#TOs	av. size	#TOs	av. size
AE	13841	11	33	3	81	4
AV	769	10	4	3	8	5
AIUC	50	5	9	2	12	3
AV-AIUC	819	10	13	3	20	4
APT	15	5	6	2	14	2

av. size = average size of the TOs

framework) and to provide statement coverage statistics (see Table 4).

5.2 Criteria Comparison

This section aims at providing a comparison of the five functional criteria proposed in Section 3.4.

For criteria comparison, we generated the corresponding test cases and measured the percentage of covered code (using the tool JTracor [30]). Statistics on the test cases generated for the three case studies are given in Table 3. For all the criteria, the average size of each test case (length of a use case sequence) varies from five to 11 instantiated use cases, and so the test cases are easy to interpret.

The results of the statement coverage measures are given in Fig. 10, in the form of the code percentage (including dead code) covered by functional test cases. Except for the *all vertices* criterion, a 70 percent statement coverage is reached by all the criteria, and combined with robustness criterion, 80 percent of the code is covered. The statement coverage of the *all vertices* criterion is weak since optimally covering all vertices of the graph does not ensure that all the use cases are used even once: the use cases that do not modify the system state are not covered (since they appear as loops on a single vertex of the UCTS).

If *all edges*, *AV-AIUC*, *all instantiated UC,* and *all precondition term* criteria are equivalent for statement coverage, this is not the case for their respective efficiency estimated in terms of a ratio between the covered statements and the test cases (see Fig. 11). Intuitively, it corresponds to the relative contribution a test case makes to statement coverage. In other words, it estimates the relative test case efficiency for covering the software. It clearly appears that test cases generated with the criteria *all edges*, *all vertices*, and *AV-AIUC* have, on average, a low efficiency. This is due to the fact that the sets of test cases generated with those criteria are not more efficient but only larger than the sets of test cases generated with the *all instantiated UC* and *all precondition terms*. Indeed, the *all edges* criterion generates, for example, 13,841 test cases (see Table 3) for virtual meeting.

TABLE 4
Statement Coverage Reached by the Generated Test Cases

	ATM	FTP	VM
% of functional code covered	100%	90.7%	100%
% of robustness wrt the spec covered	42.31%	38.6%	52%
% of code covered (total)	94.76%	72.5%	80%

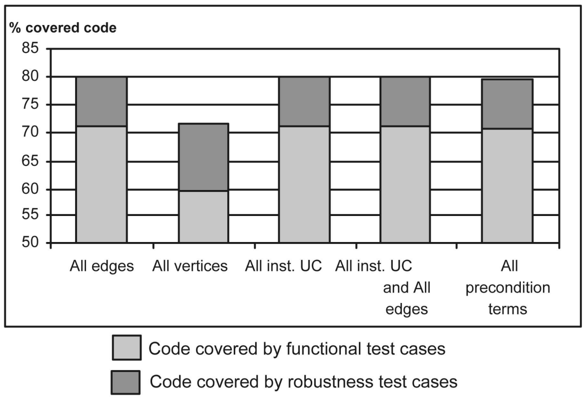


Fig. 10. A comparison of criteria with regard to statement coverage for the VM example.

As a conclusion to this study, the *all instantiated UC* and *all precondition terms* criteria appear as the most efficient for the statement coverage criterion. It is clear that this conclusion is quite sensitive to the chosen statement coverage adequacy criterion, which is somehow rough. This might explain why all criteria achieve high coverage and why more costly and complex criteria show “low efficiency.” Coverage in terms of control or data flow, or even mutants, might show different results but fall beyond the scope of this study. Yet, this study shows that our approach allows to satisfy classical industrial coverage criteria with a small number of automatically generated tests while maintaining traceability links from the requirements down to the tests cases.

5.3 Efficiency in Terms of Statement Coverage

The results of Table 4 show that the *all precondition terms* criterion is efficient to cover most of the functional code. The ratio between the number of functional test cases and the percentage of covered code is high; for instance, with only 15 test cases for the most complex study, we cover 100 percent of the functional code. This is a comforting result, and may compensate for the fact that the proportion of functional code is lower for larger systems. In comparison, the robustness criterion is quite disappointing. Indeed, less than half of the robustness code can be covered with the robustness criterion, while the number of robustness

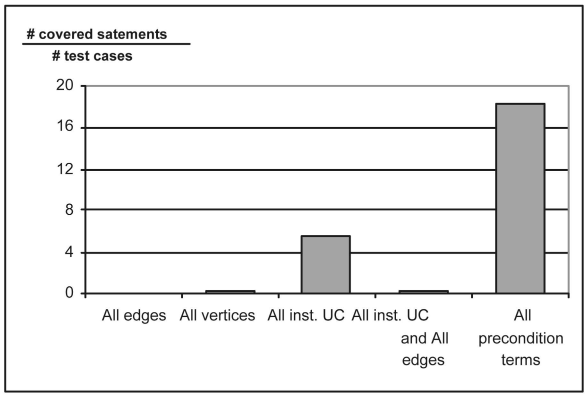


Fig. 11. A comparison of criteria with regard to test cases efficiency for the VM example.

test cases is higher than for functional testing. For example, the 65 robustness test cases only cover half of the robustness code for the virtual meeting server. Globally, if we consider that the test cases are generated directly from the requirements, in an automated way, the overall proportion of code that is covered is high (between 70 percent and 95 percent). Though promising, this proportion of covered statements is not fully satisfactory. The statement coverage, while intensively used in industry, is known to be a weak measure of the test's relevance. Being given this crude measure, the results are good concerning nominal functional code, but weak for robustness code. Testing robustness code is, in general, more difficult and requires more tricky tests than testing nominal code; thus, our robustness criterion appears as weak. The proposed approach only produces test cases able to violate the precondition terms of the use case's contracts. The robustness test cases generation would be improved by more accurately taking into account the test data, e.g., using constraint solving techniques [27]. A stronger criterion would be to generate test cases that violate precondition of the use case scenarios attached to use cases, and not only the use cases preconditions.

In conclusion, this study reveals that our approach is a useful step toward automatically achieving statement coverage based on requirements. However, this study also suggests that more efficient criteria, both for test generation and for the test efficiency measures, should be studied.

5.4 TAS Case Studies and Lessons Learned

We also applied our approach to real life case studies at Thalès. The challenge was to take the original specifications of the system written informally in natural language and see how they could be rewritten in the formalism of use case and contracts. The case studies concern two systems components of last generation combat aircrafts (Mirage 2000-9 and Rafale), of midcomplexity (several thousand C++ KLOC). The following conclusions were drawn. First, use of the simulator showed that the original definition of the system was underspecified; this was noticed since an expected transition (i.e., a possible activation of a service) was not available. This means that the corresponding requirement had not been written. The missing requirement had to be added. In that sense, the simulator allows to check if the requirements contain enough information, or are not underspecified. Second, some requirements could not be expressed in terms of use cases and contracts due to the used use case formalism (arithmetic and real-time aspects are not supported yet). Of the existing requirements, 79 percent have been successfully transformed into a set of executable use cases. The nontranslated requirements relate to detailed design features and, thus, are not taken into account when describing services requirements. Future work would consist in dealing with real-time and QoS aspects that can be expressed as high level requirements.

6 Related Work

While the need to derive functional system tests from use cases is widely recognized in the literature [31], [32], few approaches propose concrete methods to achieve this derivation. One of the cause of this lack may be that the level of detail of the use cases is coarse (and must remain coarse, as underlined in [15]), and a second one may be the fact that use cases are not as formalized as other UML views. Thus automating methods to derive tests from fuzzy descriptions of the use cases is not an easy task. The requirement-based testing techniques already existing (e.g., [33], [34], [4], [5]) are usually based on formal methods: The requirements are supposed to be written in a formal language. Conformance tests are synthesized out of these formal specifications, in general by building a behavioral automaton and covering it with some criteria. Many papers tackle the issue of test generation from formal specifications such as Z [3], B [4] or SDL [5]. Nevertheless, as identified in [35], the main drawbacks of these methods are that they are difficult to build (for the developer but even more for the clients) and to maintain over a long period. Formal methods are, thus, mainly used for very critical systems since they are far too expensive for a practical use elsewhere. In [35], it is thus suggested to focus on methods guiding the testers into a systematic test approach.

Among the numerous approaches proposed to generate tests from UML artifacts (e.g. [36], [37]), only a few them [11], [38], [39], [40], [12] are system-level and based on use cases.

Reference [38] describes an approach to generate system-level test cases from an accurate description of the use cases, including preconditions and postconditions, inspired by [8]. Each use case is transformed into a state machine, and test objectives must be defined by the tester into the underlying formalism. Criteria are also given to cover the state machines and an original application of the STRIPS AI planning formalism allows the derivation of test suites for a given statechart. The main limitation identified by the authors concerns the partial automation of the approach, e.g., the transformation from a use case description to a state machine is done manually, and the natural language is used for pre and postconditions. No experimental study is provided to estimate the efficiency of the generated test cases on the final code. The main advantage of the method presented in [38] is its speed: Due to the use of an efficient planning tool, the test cases are quickly synthesized.

In [40], a method is proposed to generate test cases in a statistical way. The base of the method relies on [38]: The use cases and their description are transformed into a state machine that is then derived into a usage graph and then a usage model, manually adding the probability distribution of the expected use of the software. Then, statistical testing is performed: Each test case is a random traversal of the usage model.

The authors of [11], [39] propose test approaches based on the dependencies between the use cases and modeling those dependencies using graphical notations.

In [39], a scenario-based approach is detailed in order to systematically derive test cases for system testing. Scenarios are formalized using state charts and a graphical notation called dependency charts, which is used to model the dependencies between scenarios. A step by step method is given to create and refine the scenarios and extract test cases using coverage criteria both on the state charts and on the dependency charts; this method is manual and its efficiency is not demonstrated. The test generation from dependency charts is more detailed in [13]. The objective of the dependency charts is larger than ours since they allow to model several kinds of dependencies such as sequential dependencies but also abstraction dependencies, time dependencies and data resource dependencies. On the other hand, test generation from such diagrams is not automated, but based on a manual application of heuristics.

Another important contribution for system testing from use cases can be found in [11], where the authors describe a complete testing method. The authors propose to express the sequential constraints of the use cases with an extended activity diagram, with new stereotypes for expressing iteration. One activity diagram is provided per actor, and the use cases are grouped into swimlanes depending on the responsibilities they have with regard to the main objects of the system. The activity diagram is then transformed into a weighted graph, from which regular expressions are exhaustively extracted. The regular expressions correspond to use case sequences. Then, all use cases are supposed to be documented with sequence diagrams, which are also transformed into regular expressions. Finally, in each sequence of use cases, the use cases are replaced with their scenario regular expressions. In this way, test cases are obtained. Our approach goes along the same lines, but differs in several important points. First, the test criterion proposed in [11] is based on the coverage of the regular expressions obtained by the projection from the activity diagram. This criterion leads to a very high number of test cases. When one is interested in a specific test adequacy criterion (such as statement coverage in our case), the problem is slightly different since the goal is to obtain the smallest number of efficient tests with regard to the chosen criterion. Second, we differ in the way the dependencies between use cases are expressed.

Like the dependency charts of [13], activity diagrams have the obvious advantage of being visual, making the dependencies between the use cases appear at a glance, while they have to be computed by a simulator with our approach. In certain cases, graphical notations may be more adequate. In other cases, contracts may be easier to handle. Thus, an interesting approach would be either to have graphical notations as a front-end to our approach, or (better) to let the requirement analyst choose the parts of the requirements she wants to model with an activity diagram, contracts, or dependency charts. We have made a first step in this direction, developing an algorithm to translate the activity diagrams proposed in [11] into contracts for the use cases [41].

In [12], the authors focus more on test data than on test control, and apply an adaptation of the category-partition method. Their approach combines a tool for test planning (cowtest) and a tool for test derivation (UID_SD). Cowtest helps to group the use cases, the sequence diagrams, and the actors in a directed dependency graph, and to associate weights to the nodes. Then IUT_SD is used to derive test scenarios according to the chosen test constraints and test strategy. Our approach has similar objectives to [12], since we also propose a global approach, integrated in the whole software life cycle, and aiming at being applied in industrial contexts.

Generating tests using declarative sequential constraints was already proposed in [42]. In this work, constraints on events are defined using temporal logic, and then test cases are generated in order to detect synchronisation faults in concurrent programs. This approach is adapted in [43] to the intraclass testing; the constraints are defined based on the pre and postconditions of the methods of a given class, and are then used for test generation using several criteria to cover the constraints. In particular, both correct and incorrect sequences of methods calls are generated (similarly, we have functional and robustness criteria). Experiments show that this approach is effective (the efficiency is measured using a mutation technique), and that the invalid sequences of method calls do not significantly improve the efficiency of a test suite. Several other approaches exist to generate tests from contracts at the operation or method level. Among them, [44] proposes to use JML (Java Modeling Language) postconditions and invariants as an oracle for unitary test cases.

In [45] the authors propose to generate tests from OCL-based specifications. The technique consists of inferring valid sequences of method calls using OCL constraints. The approach we propose uses a dedicated constraint language. Nevertheless, the proposed technique can be adapted to use OCL pre and postconditions. For that, since OCL constraints rely on a static model, a static model of the requirements needs to be extracted from the requirements. However, we believe that the OCL is not well suited for the early requirement phases because of its too complex syntax.

7 Conclusions and Future Work

Instead of pushing formal methods to the world of embedded OO software, we proposed to work the other way round, i.e., start from established practices and gently lead them toward formally exploitable models. The idea is to off-load by automation the work in test generation and to shift the effort to the specification activity. Our use case model is less sophisticated than many formal models, but it has the advantage of taking into account industrial practices and needs, such as changing requirements and time-to-market constraints and use of well-established standards (such as the UML). In this paper, we presented a complete and automated chain for test cases derivation from formalized requirements in the context of object-oriented embedded software. The underlying approach consists of improving the use cases by declarative information under the form of contracts as an anchor for further testability purposes. The test cases are generated in two steps: Use case orderings are deduced from use case contracts, and then use case scenarios are substituted for each use case to produce test cases. While in the first step the use cases model handles high level concerns, in the second step, the data complexity (numerical data, object models, OCL constraints, etc.) is taken into account with the use of use case scenarios. The proposed approach assumes the availability of a formalization of the requirements, which has to be carried out manually. The definition of this model can be quite difficult and error prone while the effectiveness of the generated tests highly depends on the quality of this model. Our experience is that in writing contracts, choosing the right set of predicates and their appropriate initial values implies going back and forth from specification to simulation and vice versa. This was not a problem in our application field since engineers in the avionics area already do spend an important effort on specification-related activities. In practice, if the specification can be provided, then the approach can be beneficial and even more so in the context of product lines, where large savings can be obtained with our automated test generation process. However, in the context of an "ordinary" engineering project, the question of whether the formalization of the requirements is worth the savings in the testing stage should be addressed before we can conclude that the overall methodology should be used in such a project.

The approach has been evaluated in three case studies by estimating the quality of the test cases generated by our prototype tools. Experiments show that most of the functional statements of the code are covered by the proposed testing technique. In fact, for the three case studies, our prototype tools generate small sets of test cases that are sufficient to cover almost all the nominal code and about half of the robustness code. Based on this admittively crude statement coverage criterion, a comparison study of the test adequacy criteria we proposed reveals that the *all precondition terms* criterion combined with the *robustness* criterion is a satisfactory trade-off between the efficiency of the obtained test set and its size.

An experimental field deployment is currently carried out in the context of collaboration (CARROLL [46]) with Thalès Airborne System (TAS). Our approach is being deployed in TAS, and tested on two avionics weapon systems. TAS experiment our prototype tool to simulate the requirements, and to generate test objectives. For confidentiality reasons, these studies cannot be detailed here, but the first results tend to confirm the scalability of the approach on real-world systems. The combinatorial explosion of the UCTS would be met if many instances of actors or business concepts are handled, and when the use cases are independent (complex interleaving are then generated). In the two real systems we studied with TAS, the use cases are tightly dependent, and the number of handled instances is low; the UCTS complexity is thus reasonable. Moreover, our prototype tool allows either the tester to define abstractions on her system so that the UCTS can be completely built or to limit the depth of the generated UCTS (indeed the test objective generator seeks to produce the shortest paths reaching a chosen system state—represented by a node in the UCTS). Additionally, an on-the-fly generation of test objectives can be performed for two of the proposed criteria. On-the-fly generation would allow us to build only part of the UCTS to generate the test objectives, and would thus make the approach suitable for systems handling more instances and independent use cases.

Our approach is being integrated with the Agatha [19] symbolic test synthesis tool because of these positive results. Since the TAS requirements are written using domain specific natural language, we are also currently developing mechanisms to fit their development processes. We defined a controlled natural language for expressing requirement. A tool is then used to parse these textual requirements and translate them into a set of use cases with contracts.

Future work will consist in extending the requirement contracts towards extra-functional properties (in particular real-time constraints), expressed into the QML QoS modeling language, and weaving these aspects to generate test cases checking these properties. Also, in this paper, we only consider the sequence diagrams to represent scenarios. In certain cases, it might be more interesting to use state machines or activity diagrams. An extension of the proposed method would consist in allowing the requirement analyst to specify her scenarios with such kind of diagrams, and to cover them in order to extract relevant linear scenarios, in order to use them to replace the use cases. Last, as evoked in related work, a good extension of the approach would be to make it compatible with graphical approaches to model the use case dependencies, such as activity diagrams.

REFERENCES

[1] P. Gibson, "Formal Requirements Models: Simulation, Validation, and Verification," technical report, Computer Science Dept., Nat'l Univ. of Maynooth, Ireland, 2001.

[2] C. Heitmeyer, J. Kirby, and B. Labaw, "Tools for Formal Specification, Verification and Validation of Requirements," *Proc. 12th Ann. Conf. Computer Assurance,* 1997.

[3] S. Helke, T. Neustupny, and T. Santen, "Automating Test Case Generation from Z Specifications with Isabelle," *ZUM '97: The Z Formal Specification Notation,* LNCS 1212, pp. 52-71. J.P. Bowen, M.G. Hinchey and D. Till, eds. Springer-Verlag, 1997.

[4] B. Legeard, F. Peureux, and M. Utting, "Automated Boundary Testing from Z and B," *Proc. Conf. Formal Methods Europe,* 2002.

[5] L. Tahat, B. Vaysburg, B. Koreland, and A. Bader, "Requirement-Based Automated Black-Box Test Generation," *Proc. 25th Ann. Int'l Computer Software and Applications Conf.,* 2001.

[6] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," *Proc. Seventh European Eng. Conf. with Seventh ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* 1999.

[7] D. D'Souza and A. Wills, "Interaction Models: Uses, Case Actions, and Collaborations," *Objects, Components, and Frameworks with UML: The Catalysis Approach,* Addison-Wesley, 1999.

[8] A. Cockburn, "Structuring Use Cases with Goals," *J. Object-Oriented Programming,* pp. 35-40, 56-62, Sept./Oct., Nov./Dec. 1997.

[9] B. Meyer, "Applying Design by Contract," *Computer,* vol. 25, no. 10, pp. 40-51, Oct. 1992.

[10] P. Kruchten, *Rational Unified Process: An Introduction.* Reading, Mass.: Addison-Wesley, 1998.

[11] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *J. Software and Systems Modeling,* pp. 10-42, 2002.

[12] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects," *Proc. Fifth Int'l Conf. Unified Modeling Language: Model Eng. Languages, Concepts, and Tools,* pp. 383-397, 2002.

[13] J. Ryser and M. Glinz, "Using Dependency Charts to Improve Scenario-Based Testing," *Proc. 17th Int'l Conf. Testing Computer Software,* June 2000.

[14] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,'' *Proc. Conf. Fourth Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 283-252, 1977.

[15] M. Fowler, "Use and Abuse Cases," *Distributed Computing,* Apr. 1998.

[16] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel, "A Requirement-Based Approach to Test Product Families," *Proc. Fifth Workshop Product Families Eng.,* 2003.

[17] C. Jard, "Synthesis of Distributed Testers from True-Concurrency Models of Reactive Systems," *Information and Software Technology,* vol. 45, no. 12, pp. 805-814, 2003.

[18] C. Jard and T. Jéron, "TGV: Theory, Principles and Algorithms," *Proc. Sixth World Conf. Integrated Design and Process Technology,* 2002.

[19] D. Lugato, C. Bigot, and Y. Valot, "Validation and Automatic Test Generation on UML Models: The AGATHA Approach," *Electronics Notes in Theorical Computer Science,* vol. 66, no. 2, 2002.

[20] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenario Usage in System Development: A Report on Current Practice," *IEEE Software,* Mar. 1998.

[21] *Recommendation ITU-TS Z.120, Message Sequence Chart (MSC),* Geneva: Int'l Telecommunication Union—Telecomm. Standardization Sector, 1999.

[22] "Unified Modeling Language: OCL," Object Management Group, http://www.omg.org/docs/ptc/03-08-08.pdf, 2005.

[23] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

[24] E. Gamma and K. Beck, "JUnit," http://junit.org, 2005.

[25] L. Briand, W. Dzidek, and Y. Labiche, "Using Aspect-Oriented Programming to Instrument OCL Contracts in Java," Technical Report SCE-04-03, Carleton Univ., 2004.

[26] *OCL Toolkit,* http://dresden-ocl.sourceforge.net, 2005.

[27] K. Marriott and P.J. Stuckey, *Programming with Constraints.* MIT Press, 2000.

[28] R. Bjork, "An ATM Simulation," http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample, 2005.

[29] F. Cueto, http://cqs.dyndns.org:81/javaftp, 2005.

[30] F. Fleurey, "A Framework to Trace Execution of Java Programs," http://franck.fleurey.free.fr/jtracor, 2005.

[31] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Softaware Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[32] R. Binder, *Testing Object-Oriented Systems.* Addison-Wesley, 2000.

[33] G. Bernot, M.-C. Gaudeland, B. Marre, "Software Testing Based on Formal Specifications: A Theory and a Tool," *Software Eng. J.,* vol. 6, no. 6, pp. 387-405, 1991.

[34] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Modelbased Specifications," *Proc. Int'l Symp. Formal Methods Europe,* pp. 268-284, 1993.

[35] J. Ryser, S. Berner, and M. Glinz, "On the State of the Art in Requuirements-Based Validation and Test of Software," technical report, Inst. Für Informatik, Univ. of Zurich, 1998.

[36] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," *Proc. Second Int'l Conf. Unified Modeling Language: Beyond the Standard,* 1999.

[37] Y. Kim, H. Honh, S. Cho, D. Bae, and S. Cha, "Test Cases Generation from UML State Diagrams," *IEE Proc. Software,* vol. 146, no. 4, pp. 187-192, Aug. 1999.

[38] P. Fröhlich and J. Link, "Automated Test Case Generation from Dynamic Models," *Proc. 14th European Conf. Object-Oriented Programming,* 2000.

[39] J. Ryser and M. Glinz, "A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts," *Proc. 12th Int'l Conf. Software and Systems Eng. and Their Applications,* Dec. 1999.

[40] M. Riebisch, I. Philippow, and M. Götze, "UML-Based Statistical Test Case Generation," *Proc. Int'l Conf. Net.ObjectDays,* vol. 2591, pp. 394-411, 2002.

[41] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel, "Requirements by Contracts Allow Automated System Testing," *Proc. 14th IEEE Int'l Symp. Software Reliability Eng.,* 2003.

[42] R.H. Carver and K.-C. Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 471-490, June 1998.

[43] F. Daniels and K. Tai, "Measuring the Effectiveness of Method Test Sequences Derived from Sequencing Constraints," *Proc. Technology of Object-Oriented Languages and Systems,* pp. 74-83, 1999.

[44] Y. Cheon and G.T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," Technical Report 01-12, 2001, http://citeseer.nj.nec.com/cheon01simple.html.

[45] M. Benattou, J.-M. Brueland, N. Hameurlain, "Generating Test Data from OCL Specification," *Proc. ECOOP Workshop Integration and Transformation of UML Models,* 2002.

[46] "Programme de Recherche CARROLL," Thalès, INRIA, and CEA, http://www.carroll-research.org, 2005.

Clémentine Nebut received the engineering degree and the PhD in computer science from the University of Rennes 1, France. She is an associate professor at the University of Montpellier 2, France. Her research interests are software testing and model-driven engineering.

Franck Fleurey is a PhD student on the Triskell project team at the University of Rennes. His research interests are model-driven engineering and software validation.

Yves Le Traon received the engineering degree and the PhD degree in computer science from the Institut National Polytechnique de Grenoble, France. He is a research engineer at France Télécom Research and Development, and an associate member of the IRISA research laboratory. His research interests include OO testing, design for testability, and software measurement. He is a member of the IEEE.

Jean-Marc Jézéquel received the PhD degree in computer science from the University of Rennes, France, in 1989. He joined the Centre National de la Recherche Scientifique in 1991. Since October 2000, he has been a professor at the University of Rennes, leading an INRIA research team called Triskell. His interests include model-driven software engineering based on object-oriented technologies for telecommunications and distributed systems. He is a member of the IEEE.