Coverage Criteria for Model-Based Testing using Property Patterns

Kalou Cabrera Castillos
LAAS-CNRS
BP 54200
31031 Toulouse cedex 4, France
kcabrera@laas.fr

Frédéric Dadeau   Jacques Julliand
FEMTO-ST Institute/INRIA CASSIS Project
16 route de Gray
25030 Besançon cedex, France
frederic.dadeau@femto-st.fr   jacques.julliand@femto-st.fr

We present in this paper a model-based testing approach aiming at generating test cases from a UML/OCL model and a given test property. The property is expressed using a dedicated formalism based on patterns, and automatically translated into an automaton. We propose new automata coverage criteria that are tailored to the property automata we consider. These criteria are based on the coverage of a relevant subset of the transitions related to the original property, aiming at producing test cases that illustrate the dynamics of the system described in the property. In addition, we propose to produce test cases that can ensure the robustness of the system w.r.t. the property, by mutating the property automaton, in order to try to provoke events that would be forbidden by the property. This approach has been implemented into a large tool set and it has been experimented on realistic case studies, in the context of industrial research projects.

Keywords. Model-Based Testing, UML/OCL, property patterns, coverage criteria, automata mutation.

1 Introduction and Motivations

Model-Based Testing is one of the best ways to automate software testing. Even if the design of the model is a costly task, it is damped by the profits that can be made from it. Indeed, the model can be used to automatically compute the test cases, from simple input test data to complete sequences involving several steps. In addition, it is used to provide the oracle, namely the expected results of the test, thus deciding of the conformance between the system under test and its formal representation. Many model-based software testing tools exist, such as Smartesting's CertifyIt [8], Conformiq Designer [17], or STG [9].

In practice, some of these tools work by automatically applying a structural test selection criterion on the model (such as the structural coverage of the OCL code [23] of the operations contained in an UML class diagram). Even if this approach is quite effective in practice, it suffers from its subjectivity and thus, specific behaviours of the system, which require a more intensive test effort, are not much targeted by this testing strategy. To overcome this problem, dynamic test selection criteria are introduced. They consist in scenario-based testing approaches that aim at exercising more extensively specific parts of the considered system.

In general, test scenarios are expressed in a dedicated language that can be either textual [5] or graphical [9], describing sequences of steps (usually operation calls) that can be performed, along with possible intermediate states reached during the unfolding of the scenario. Nevertheless, the design of the test scenarios remains a manual task that we aim to automate. During previous experiments in the use of scenarios, we have noticed that scenarios often originate from a manual interpretation of a given property that exercises the dynamics of the system [18]. Our goal is now to express such properties, in a simple formalism, that can be later exploited for testing purposes. To achieve that, Dwyer *et al.* introduced

A. Petrenko, H. Schlingloff (Eds.): Ninth Workshop on
Model-Based Testing (MBT 2014)
EPTCS 141, 2014, pp. 29–43, doi:10.4204/EPTCS.141.3

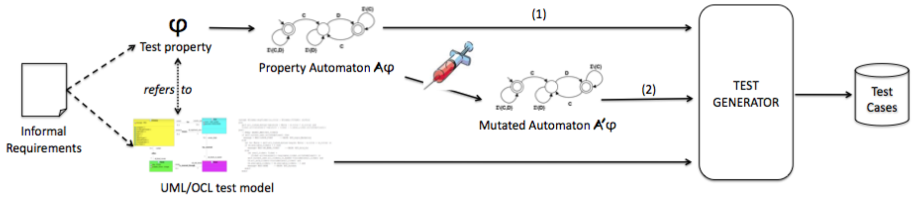


Figure 1: Process of the property-based testing approach

the notion of *property patterns* that makes it possible to express dynamic behaviours of the systems without employing complex temporal logics formalism [12]. Such properties are expressed with a *scope*, that delimits some fragments of the execution of the system, and a *pattern* that expresses occurrences, absences or chains of given events inside the considered scope.

The proposed process is depicted in Fig. 1. A test property φ is described using a property pattern that holds on the UML/OCL model. This property is translated into an automaton $\mathscr{A}_\varphi$, which captures the events that are described in the test property. Such an automaton represents a monitor of the satisfaction of the property during the execution of the system/model. It can be used to measure the coverage of the property by existing test suites, as explained in [7], but also to produce test cases that aim at exercising the property according to dedicated coverage criteria. We introduce, in a first contribution, dedicated property automata coverage criteria, inspired from classical automata coverage criteria, that aim at characterizing relevant tests highlighting the behaviours described in the property. In this context, the tests are directly extracted from the automaton as is (arrow 1), to cover the events of the property. As a second contribution, we propose to refine the property automaton so as to exhibit possible violations of the property, and derive test cases that aim at ensuring the robustness of the system w.r.t. the property. To this end, we apply dedicated mutation operators (arrow 2) on the transitions of the original automaton $\mathscr{A}_\varphi$, producing a new automaton $\mathscr{A'}_\varphi$ that is then exploited similarly.

This paper is organized as follows. Section 2 presents the considered UML subset that is used for our model-based testing approach and introduces a running example. Then, Section 3 presents the property formalization language and its associated semantics as automata. The first contribution, namely the property automata coverage criteria, is presented in Section 4. The second contribution, namely the mutation of automata transitions, is explained in Section 5. The experimental assessment of this approach is reported in Section 6. Then, Section 7 compares our approach with related works. Finally, Section 8 concludes and presents the future works.

2 UML/OCL Models and Running Example

This section presents the UML/OCL models that are considered in our approach, along with a running example that will be used to illustrate the contributions in the rest of the paper. As our approach is technically bound to the CertifyIt test generation engine, we focus on the subset of UML considered by this tool. However, the approach could be applicable to any other modelling language.

2.1 UML4ST – a subset of UML for Model-Based Testing

The UML models we consider are those supported by the CertifyIt test generator, commercialized by the Smartesting company. This tool automatically produces model-based tests from a UML model [4] with OCL code describing the behaviors of the operations. CertifyIt does not consider the whole UML notation as input, it relies on a subset named UML4ST (UML for Smartesting) which considers class diagrams, to represent the data model, augmented with OCL constraints [23], to describe the dynamics of the system. It also requires the initial state of the system to be represented by an object diagram. Finally, a statechart diagram can be used to complete the description of the system dynamics.

OCL provides the ability to navigate the model, select collections of objects and manipulate them with universal/existential quantifiers to build first-order logic expressions. Regarding the OCL semantics, UML4ST does not consider the third logical value *undefined* that is part of the classical OCL semantics. All expressions have to be defined at run time in order to be evaluated. CertifyIt interprets OCL expressions with a strict semantics, and raises execution errors when encountering null pointers.

These restrictions w.r.t. the classical UML semantics originate from the fact that the UML/OCL model aims at being used for Model-Based Testing purposes. As such, it requires to use an executable UML/OCL model, since the abstract test cases are obtained by animating the model.

2.2 Running Example

We illustrate the UML/OCL models that are considered using a simple model of a web application named eCinema. This application provides a means for registered users to book tickets for movies that are screened in a cinema.

The UML class diagram, depicted in Fig. 2 contains the classes of the application: `ECinema`, `Movie`, `Ticket` and `User`. The `ECinema` class models the system under test (SUT) and contains the API operations offered by the application. This application proposes classical online booking features: a registered user may login to application, purchase tickets, view his basket, delete one or all tickets from his basket, and logout.

Figure 3 shows the OCL code of the *buyTicket* operation. Upon invocation, the caller provides the title of the movie for which (s)he wants to buy a ticket. The operation first checks that the user is registered on the application, and then checks if there exists an unallocated tickets for this movie. If all these verifications succeed, a ticket is associated to the user. This operation is specified in a defensive style, and as such, its precondition is always true, and the postcondition is in charge of distinguishing nominal cases from erroneous calls. We assume in the rest of the paper that all operations are specified this way, which is realistic since the model is used for testing purposes.

The OCL code of this operation contains non-OCL annotations, inserted as comments, such as `---@AIM: id` and `---@REQ: id`. The `---@AIM:id` tags denote test targets while the `---@REQ: id` tags mark requirements from the informal specifications. These tags can be used to reference a particu-

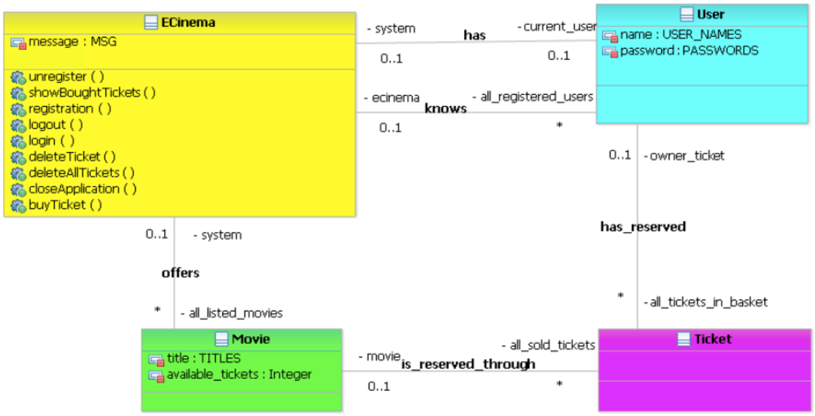


Figure 2: Class diagram of the eCinema model

```
context ECinema::buyTicket(in_title : ECinema::TITLES): oclVoid
effect:
   ---@REQ: BASKET_MNGT/BUY_TICKETS
   if self.current_user.oclIsUndefined() then
     message = MSG::LOGIN_FIRST      ---@AIM: BUY_Login_Mandatory
   else
      let tm: Movie = self.all_listed_movies->any(m: Movie | m.title = in_title) in
      if tm.available_tickets = 0 then
         message= MSG::NO_MORE_TICKET     ---@AIM: BUY_Sold_Out
      else
         let t: Ticket = (Ticket.allInstances())->any(owner_ticket.oclIsUndefined()) in
         self.current_user.all_tickets_in_basket->includes(t) and
         tm.all_sold_tickets->includes(t) and
         tm.available_tickets = tm.available_tickets - 1 and
         message= MSG::NONE     ---@AIM: BUY_Success
      endif
   endif
```

Figure 3: OCL code of the `buyTicket` operation

lar behaviour of the operation (e.g. `@AIM:BUY_Login_Mandatory` represents a failed invocation of this operation, due to the absence of a user logged on the system). Notice that it is possible to know which tags were covered during the execution of the model, inside the test cases, providing a feedback on the structural coverage of the OCL by the test cases.

3 Property Patterns and Automata

We now present the property language and their associated property automata.

3.1 Property Pattern Language

The property description language is a temporal extension of OCL. This language is based on patterns which avoids the use of complex temporal formalisms, such as LTL or CTL. We ground our work on the initial proposal of Dwyer *et al.* [12] in which a temporal property is a *temporal pattern* that holds within a *scope*. Thus, the user can define a temporal property choosing a pattern and a scope among a list of predefined schema. The scopes are defined from *events* and delimit the impact of the pattern. The patterns are defined from events and *state properties* to define the execution sequences that are correct. The state properties and the event are described based on OCL expressions.

Patterns. There are five main temporal patterns: (i) `always` P means that state property P is satisfied by any state, (ii) `never` E means that event E never occurs, (iii) `eventually` E means that event E eventually occurs in a subsequent state, this pattern can be suffixed by a bound which specifies how many occurrences are expected (at least k, at most k, exactly k times), (iv) E_1 (`directly`) `precedes` E_2 means that event E_1 (directly) precedes event E_2, (v) E_1 (`directly`) `follows` E_2 means that event E_2 is (directly) followed by event E_1.

Scopes. Five scopes can apply to a temporal pattern TP: (i) TP `globally` means that TP must be satisfied on any state of the whole execution, (ii) TP `before` E means that TP must be satisfied before the first occurrence of E, (iii) TP `after` E means that TP must be satisfied after the first occurrence of E, (iv) TP `between` E_1 `and` E_2 means that TP must be satisfied between any occurrence of E_1 followed by an occurrence of E_2, (v) TP `after` E_1 `until` E_2 means that TP must be satisfied between any occurrence of E_1 followed by an occurrence of E_2 and after the last occurrence of E_1 that is not followed by an occurrence of E_2.

Events. Scopes and patterns refer to events that can be of two kinds. On the one hand, events denoted by `isCalled(op, pre, post, {tags})` represent operation calls. In this expression, `op` designates the operation name, `pre` and `post` are OCL expressions respectively representing a precondition and a postcondition. Finally, `{tags}` represents a set of tags that can be activated by the operation call. Such an event is satisfied on a transition when the operation `op` is called from a source state satisfying the precondition `pre` and leading to a target state satisfying the postcondition `post` and executing a path of the control flow graph of the operation `op` which is marked by at least one tag of the set of tags denoted `{tags}`. Notice that the three components `pre`, `post` and `{tags}` are optional. On the other hand, events denoted by `becomesTrue(`P`)` where P is a OCL predicate, are satisfied by any operation call from a state in which P evaluated to false, reaching a state in which P evaluates to true.

Example 1 (Property Example) *Consider the eCinema example given in Sect. 2.2. An informal access control requirement expresses that: "the user must be logged on the system to buy tickets". This can be expressed by the following three properties that put the focus on various parts of the execution of the system.*

never *isCalled(buyTicket, {@AIM:BUY_Success})*
before *isCalled(login, {@AIM:LOG_Success})* (*Property* 1)

eventually *isCalled(buyTicket, {@AIM:BUY_Success})* ***at least 0 times***
between *isCalled(login, {@AIM:LOG_Success})* (*Property* 2)
and *isCalled(logout, {@AIM:LOG_Logout})*

never *isCalled(buyTicket, {@AIM:BUY_Success})*
after *isCalled(logout, {@AIM:LOG_Logout})* (*Property* 3)
until *isCalled(login, {@AIM:LOG_Success})*

First, with Property (1)*, we specify that before a first successful login, it is not possible to succeed in buying a ticket. Second, we specify that when the user is logged in, he may buy a ticket (Property* (2)*). Notice that this property uses a workaround of the eventually pattern to express an optional action. Finally, Property* (3) *specifies that it is also impossible to buy a ticket after logging out, unless logging in again.*

3.2 Property Semantics using Automata

The properties are interpreted on executions that are viewed as sequences of pairs of a state and an event that represent a sequence of transitions. The semantics of the test properties are expressed by means of automata. Indeed, the temporal language is a linear temporal logic whose expression power is included in the ω-regular languages.

The semantics of a temporal property is a labelled automaton which is defined by Def. 1. The method that associates an automaton to a temporal property is completely defined in [6]. This automaton describes the set of accepted executions of the property and highlights specific transitions representing the events used in the property description. In addition, the automaton may contain at most one rejection state that indicates the violation of the property when reached.

Definition 1 (Property Automaton) *Let* Σ *be a set of events. An automaton is a quintuplet* $\mathscr{A} = \langle Q, q_0, F, R, T\rangle$ *in which:* Q *is a finite set of states,* q_0 *is an initial state* ($q_0 \in Q$)*,* F *is a set of final states* ($F \subseteq Q$)*,* R *is a set of rejection states* ($R \subseteq Q$)*,* T *is a set of transitions* ($T \subseteq Q \times \mathscr{P}(\Sigma) \times Q$) *labelled by a set of events.*

We call $\alpha-$transitions the transitions of T that are labelled by the events expressed in the original property, and we call $\Sigma-$transitions the other transitions. $\Sigma-$transitions are named after their expression as they are labelled by a restriction on Σ (the set of all possible events).

Notice that, when considering safety properties (something bad never happens), the set R of rejection state is necessarily not empty. Notice also that, for a given state (resp. transition), it is possible to know if the state (resp. transition) originates from the scope or the pattern of the property. Notice also that the final states catch that the scope has been executed at least once. Thus final states are not accepting states as in traditional Büchi automata; they represent the test goals in the sense that we expect test cases to reach such states at some point.

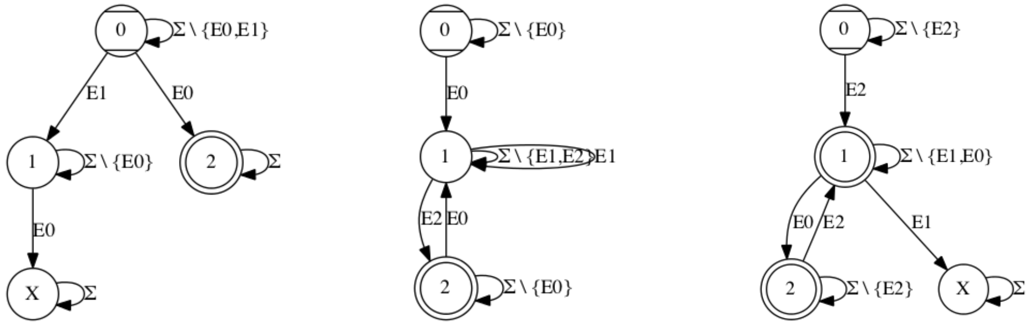
Events in the automaton are quadruplets $[op, pre, post, \{tags\}]$ in which *op* designates an operation, *pre* and *post* respectively denote pre- and postconditions, and *tags* specifies a set of tags. The events used in the test properties are thus rewritten to match this formalism: `isCalled(`op, pre, post, $\{$tags$\}$`)` rewrites to $[op, pre, post, \{tags\}]$ and `becomesTrue(`P`)` rewrites to $[\_, not(P), P, \_]$, in which $\_$ replaces any acceptable value of the corresponding component.

Example 2 (Automaton of a Temporal Property) *Consider the property given in Example 1. Figure 4 shows the automata representation associated to Property* 1 *(left), Property* 2 *(middle) and Property* 3 *(right). Notice that the left-hand side (resp. right-hand side) automaton displays an error state, identified by "X", that can be reached if the system authorizes to perform a ticket purchase before logging in (resp. after logging out and before logging in again). The automaton of Property 2, in the middle, does not display an error state meaning that the property can never be falsified. In addition, it exhibits a reflexive transition that represents the optional event, that may or may not take place when the user is logged.*

On these automata, the $\alpha-$transitions are the transitions labelled by events E0, E1 and E2. The other transitions are $\Sigma-$transitions.

These automata provide a means for monitoring the satisfaction of the property by the execution of the model. We assume that the model is correct w.r.t. the property. In that sense, only valid traces can be extracted from the model, and no transition leading to an error state can possibly be activated.

The next two sections show how to exploit these automata by defining dedicated test coverage criteria, that can be used either for evaluating an existing test suite, or for generating new tests supposed to



Property 1 Property 2 Property 3

E0: [login,_,_,{@AIM:LOG_Success}]
E1: [buyticket,_,_,{@AIM:BUY_Success}]
E2: [logout,_,_,{@AIM:LOG_Logout}]

Figure 4: Automata representation for the properties given in Example 1

illustrate or exercise the property.

4 Property Automata Coverage Criteria

We present in this section the automata coverage criteria that we propose. These dedicated coverage criteria originate from the observation that classical coverage criteria on automata are not relevant for our property automata. Indeed, criteria such as transition coverage, or transition-pair coverage, make no distinction between the transitions of the automata. However, in our case, all the transitions of the property automata are not of equal importance. For example, consider the automata provided in Fig. 4. Reflexive $\Sigma-$transitions only exist to capture all possible executions of the model but their sole purpose is to make it possible to put the model in a state from which the $\alpha-$transitions can be activated. While classical coverage criteria would target these Σ-transitions, we propose new coverage criteria focused on $\alpha-$transitions, aiming at activating them, but also focusing on different paths which iterate over specific parts of the automaton. We present these criteria and illustrate, for each of them, their relevance in terms of property coverage. Before that, we start by introducing some preliminary definitions.

4.1 Preliminary Definitions

We consider that an abstract test case is defined on the model as a finite sequence of steps, each of them formalized by $s_{i+1}, \vec{o_i}, tags_i \leftarrow op_i(\vec{in_i}, s_i)$ (for $i \geq 0$ and $i <$ the length of the test case) in which s_i (resp. s_{i+1}) is the model state before (resp. after) the step, op_i is a model operation called with inputs $\vec{in_i}$ returning outputs $\vec{o_i}$ and activating the behaviours identified by the $tags_i$ set. We denote by s_0 the initial state of the model.

The conversion of a test case (computed from the model) into a path of the automaton is made by matching the steps of the test case with the events of the automaton, accordingly to the following definition.

Definition 2 (Step/Event matching) *A step formalized by $s_{i+1}, \vec{o_i}, tags_i \leftarrow op_i(\vec{in_i}, s_i)$ is said to match an event $[op, pre, post, tags]$ if and only if the four conditions hold: (i) $op = op_i$ or op is undefined (symbol _), (ii) pre is satisfied in s_i (modulo substitution of $\vec{in_i}$ in pre), (iii) post is satisfied in s_{i+1}, and (iv) $tags \cap tags_i \neq \emptyset$*

Given a test case, each step $s_{i+1}, \vec{o_i}, tags_i \leftarrow op_i(\vec{in_i}, s_i)$ is matched against the possible transitions $q \xrightarrow{e_i} q'$ that can be activated from the current automaton state q (initially, q_0 when the first step is considered). When a given step/event is matched, the exploration of the automaton restarts from q' the state targeted by the transition. As the property automata are deterministic and complete, there is exactly one transition that can be matched by each step of the test case.

4.2 Coverage Criteria for the Property Automata

We present in this section the four coverage criteria that we propose. Since the model is expected to satisfy the property, the paths inescapably leading to the error state are not supposed to be coverable, and thus, their transitions are not considered in the coverage criteria that we now present.

The first two coverage criteria that we propose consider the α-transitions.

Definition 3 ($\alpha-$transition coverage) *A test suite is said to satisfy the $\alpha-$transition coverage criterion if and only if each $\alpha-$transition of the automaton is covered by at least one test case of the test suite.*

This first coverage criterion is an adaptation of the classical transitions-coverage criteria from the literature [16]. It aims at covering the transitions that are labelled by events initially written in the associated temporal property. A test suite satisfying this criterion ensures that all the events expressed at the property level are highlighted by the test suite.

Example 3 ($\alpha-$transition coverage) *On the example shown in Fig. 4, for Property 2, a test suite satisfying the $\alpha-$transition coverage criterion ensures that at least one test case illustrates the optional ticket purchase by covering transition $1 \overset{E1}{\to} 1$. Also, another test case should illustrate the fact that two iterations of the scope are possible, by covering transition $2 \overset{E0}{\to} 1$.*

Definition 4 ($\alpha-$transition-pair coverage) *A test suite is said to satisfy the $\alpha-$transition-pair coverage criterion if and only if each successive pair of $\alpha-$transitions is covered by at least one test case of the test suite.*

Notice that this criterion considers the coverage of pairs of $\alpha-$transitions reaching a particular state, and originating from the same state. However, it is possible to display intermediate $\Sigma-$transitions between a pair of $\alpha-$transitions.

Example 4 ($\alpha-$transition-pair coverage) *On the example shown in Fig. 4, for Property 2, a test suite satisfying the $\alpha-$transition coverage criterion ensures the coverage of the following pairs: $(0 \overset{E0}{\to} 1, 1 \overset{E1}{\to} 1)$, $(0 \overset{E0}{\to} 1, 1 \overset{E2}{\to} 2)$, $(1 \overset{E1}{\to} 1, 1 \overset{E2}{\to} 2)$, $(1 \overset{E2}{\to} 2, 2 \overset{E0}{\to} 1)$, $(2 \overset{E0}{\to} 1, 1 \overset{E1}{\to} 1)$ and $(2 \overset{E0}{\to} 1, 1 \overset{E2}{\to} 2)$. A test suite satisfying this coverage criterion thus ensure the existence of tests illustrating the buying of a ticket, but also tests performing a login followed by a logout without intermediate ticket purchase, and also tests illustrating the optional ticket purchase in a second iteration over the scope.*

The last two coverage criteria that we propose consider the structure of the property and aim at covering internal or external loops inside the property automaton, in order to iterate over the pattern or the scope of the property.

Definition 5 (k-pattern coverage) *A test suite is said to satisfy the k-pattern-activation coverage criterion if and only if the $\alpha-$transitions of the pattern of the automaton are iterated between 0 and k times, each loop in the pattern being performed without exiting the pattern-part of the automaton.*

This coverage criterion aims at activating the internal loops inside the pattern-part of the automaton, without covering any transition of scope during these iterations. This coverage criterion is not applicable to any pattern; it only applies to `precedes`, `follows` and some forms of the `eventually` pattern.

Example 5 (k-pattern-coverage) *On the example shown in Fig. 4, for Property 2, a test suite satisfying the 2-pattern coverage criterion ensures the coverage of 0, 1, and 2 iterations of the reflexive $\alpha-$transition $1 \overset{E1}{\to} 1$.*

Definition 6 (k-scope coverage) *A test suite is said to satisfy the k-scope-activation coverage criterion if and only if the $\alpha-$transitions of the scope of the automaton are iterated between 1 and k times, and covering each time at least one $\alpha-$transition of the pattern.*

This coverage criterion aims at activating the external loops outside the pattern-part of the automaton. Similarly to the k-pattern criterion, the k-scope criterion is not applicable to all scopes, but restricts its usage to only repeatable ones, namely `between` and `after`.

Example 6 (k-scope coverage) *On the example shown in Fig. 4, for Property 2, a test suite satisfying the* 2*-scope coverage criterion ensures the coverage of 1 and 2 logout-login sequences, by covering cycle* $1 \overset{E1}{\rightarrow} 2 \overset{E0}{\rightarrow} 1$.

These four coverage criteria are based on the property automata as is, and thus, will only illustrate the property and show that they are correctly implemented (e.g. the occurrences of events are authorized by the implementation, along with the repetition of scopes, etc.) However, showing that unexpected events do not appear requires an additional and dedicated strategy for robustness testing, that we now present.

5 Property Automata Mutation for Testing Robustness

The automata coverage criteria described in the previous section focus on activating events expressed within the test properties. Thus, these coverage criteria aim at illustrating that the properties are correctly implemented. However, in the cases of safety properties (something bad should never happen), it might be interesting to produce test cases that aim at an attempt to violate the property. The cases in which the property is violated are clearly identified in the automaton, being displayed through error states. Unfortunately, targeting the activation of transitions leading to these error states is irrelevant: since the model (used to compute the tests) is supposed to satisfy the property, these transitions can not be activated as is.

In this section, we propose mutation operators that apply to events labelling the transition leading to error states, so as to make them activable, thus providing an interesting test target that aims at leading to a possible violation of the property. These mutations and the robustness coverage criterion aims to simulate an erroneous implementation of the property that would mistakenly allow the activation of a unexpected event. As we are performing a Model-Based Testing approach, it is thus mandatory to be able to compute a sequence performing forbidden events, at the model level.

5.1 Event Mutation Operators

Our goal is to provoke unexpected events. As these latter can not be activated on the model, the idea is to get closer to the inactivable event. To achieve that, we apply mutations on these events. These mutations apply mainly to the uncontrollable part of the events (postconditions and tags), and keep the controllable part the lesser modified.

The mutations we propose modify the transitions of the automata. They target the events labelling the transitions, and can be of two kinds: (i) predicate mutation rules, inspired from classical mutation operators over predicates [11], applied to pre- and postconditions, and (ii) tag mutation rules applied to the tag list of the events.

Postcondition/Tag Removal. This rule consists in removing the postcondition and the tag list from the event.

$$[op, pre, post, T] \rightsquigarrow [op, pre, \_, \_]$$

Both tags and postconditions are systematically removed, as these two elements are frequently related. Their combined removal thus avoids creating inactivable events.

Precondition Removal. This rule consists in removing the precondition of the event.

$$[op, pre, post, T] \rightsquigarrow [op, \_, \_, \_]$$

When applied, this mutation also removes the postcondition and tags, in order to weaken the event, and increase the chances that the mutation will produce an activable event.

Predicate Weakening. The predicate removal mutation replaces each literal in a conjunction by true. This removal applies to both pre- and postconditions.

$$[op, A \wedge B, C \wedge D, T] \rightsquigarrow [op, A, \_, \_], [op, B, \_, \_], [op, A \wedge B, C, \_], [op, A \wedge B, D, \_]$$

When applied to the postcondition, this rule removes the tags from the event. If it is applied to the precondition, this rule also removes the postcondition from the event.

Example 7 (Event mutation) *Consider the examples provided on Fig. 4, left-hand side or right-hand side. In both cases, event* $E1 = [buyticket, \_, \_, \{@AIM : BUY\_Success\}]$ *can be rewritten to* $E'1 = [buyticket, \_, \_, \_]$. *This event represents the attempt to perform a ticket purchase but without any expectation regarding the success or the failure of this operation.*

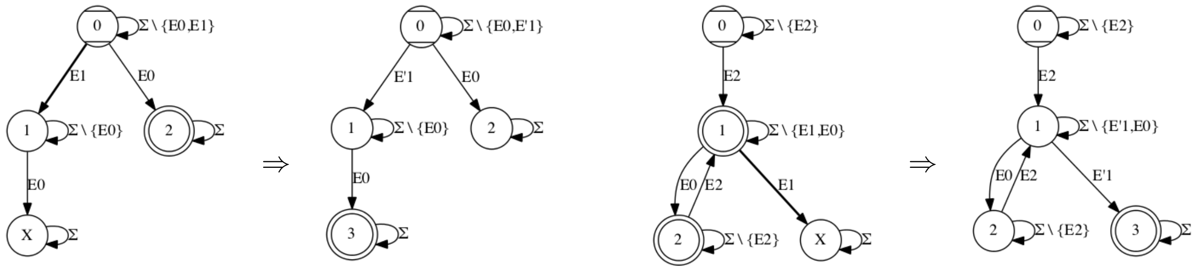
5.2 Automata Mutation and Robustness Coverage Criteria

The mutation operators that we propose can be applied on a given property automaton $\mathscr{A}$. The automaton is modified as follows: (i) each transition leading to the error state is mutated, and (ii) the targeted error state becomes the only final state of the new automaton. We denote $\mathscr{A}'$ the new automaton obtained after mutation.

Example 8 (Automaton mutation) *Figure 5 displays the application of a mutation on the automaton associated to Properties* (1) *and* (3). *We see that the mutated automaton makes it possible to match test cases that would perform an attempt to purchase a ticket, before successfully logging in.*

Definition 7 (Robustness coverage) *A test suite is said to satisfy the robustness coverage criterion for a property P if and only if the mutated transition of each mutated automaton of property P is covered by at least one test case of the test suite.*

Example 9 (Robustness coverage) *In order to activate the mutated event of Property 1, and thus, check the robustness of the system w.r.t. it, the validation engineer can design the following test case:*



E1: [buyticket,_,_,{@AIM:BUY_Success}] $\rightsquigarrow$ E'1: [buyticket,_,_,_]

Figure 5: Mutation of the automaton for Property 1 (left) and Property 3 (right)

Step	Operation	Expected behavior
1	`sut.buyTicket(TITLE1)`	`@AIM:BUY_Error_Login_First`
2	`sut.login(REGISTERED_USER,REGISTERED_PWD)`	`@AIM:LOG_Success`

On a correct implementation, the system should not allow the first operation (`buyTicket`*) to be performed successfully. If the implementation conforms to the model, then it is expected to activate an erroneous behavior of this operation (as predicted by the model). If the implementation is incorrect, the* `buyTicket` *operation will return successfully and thus display a behavior that differs from the model.*

In order to activate the mutated event of Property 3, the validation engineer can design the following test case:

Step	Operation	Expected behavior
1	`sut.login(REGISTERED_USER,REGISTERED_PWD)`	`@AIM:LOG_Success`
2	`sut.logout()`	`@AIM:LOG_Logout`
3	`sut.buyTicket(TITLE1)`	`@AIM:BUY_Error_Login_First`

Similarly, on a correct implementation, the last operation (`buyTicket`*) should not succeed (as on the model). An incorrect implementation would allow this operation be performed successfully.*

We now present the experimental assessment of the approach proposed in this paper.

6 Experimentation

Our approach has been implemented into a specific framework [10], and applied in an industrial context during two national projects. This framework allows the user to write properties, measure the coverage of a property with an existing test suite, and generate test scenarios to satisfy a given (selected) coverage criterion[1]. We report here the usage of the tool and its evaluation by test engineers. In addition, we present an experimental evaluation of the capabilities of our tool in terms of fault detection, especially focusing on the robustness testing criteria.

Experimentations during industrial projects. Our approach has been initially designed during the ANR TASCCC project[2], and implemented into a prototype of the same name. This project was done in collaboration with (among others) the Smartesting company (technology provider with the CertifyIt test generator), and Gemalto (case study provider) and focused on the validation of smart card products for Common Criteria evaluations. In this context, the properties we designed aimed at expressing functional security properties on a smart card case study. It has also been exploited during the ANR OSeP project[3], also in partnership with the Smartesting company, and funded by the Armaments Procurements Agency. This project focused on online and offline techniques for testing security (cryptographic) components. In both projects, we had the opportunity to address the following three questions: *How easy it is to learn and use the test property language?* $(Q.\ 1)$ *How does this approach compare to a functional approach such as Smartesting CertifyIt?* $(Q.\ 2)$ *What is the interest of our industrial partners in our property-based testing approach?* $(Q.\ 3)$

For our experiments, we considered the case studies developed by our partners. We started by designing test properties for the three considered case studies. The language turned out to be easy to learn, as the number of combinations is relatively small. With a little help from us, the validation engineers were

[1] see `http://vimeo.com/53210102` for a video demo of the tool

[2] `http://lifc.univ-fcomte.fr/TASCCC`

[3] `http://osep.univ-fcomte.fr`

able to express some security properties in natural language, rephrase them in order to fit the allowed combinations of scopes/patterns, and choose the appropriate entities to express these properties [Q. 1]. However, we noticed some pitfalls in its use. Indeed, the validation engineers tend to express test cases with the properties, e.g. by specifying sequences of events without taking care of the applicability of the sequence in general.

For each case study, our industrial partners generated functional test cases using the Smartesting CertifyIt test generator. Thus, we measured the coverage of the test properties that were designed. For each of the three considered case studies, we noticed that some properties where not covered at all (no $\alpha-$transition of their associated automaton were activated), meaning that the test cases do not activate the scope of the property [Q. 2]. Such a process provides an interesting feedback on the coherence of the model w.r.t. the properties and the completeness of the test suite. When a given coverage criterion is not satisfied, the validation engineer has the possibility to see which parts of the automaton were not covered, and focus his effort on them [Q. 3]. In some cases, we detected some violations of the property during the test cases executions. In this case, the coverage report shows that transitions leading to the error state of the property automaton is reached. This indicates that the model does not satisfy the property. There are two reasons for that, either the property is too restrictive (e.g. in the cases described above), or the model is incorrectly designed (this was never the case). It is then mandatory to correct one or the other of these artefacts. In this end, this process can be (indirectly) used to validate the model, by checking that it behaves as described in the properties.

In addition, we asked our industrial partners to evaluate the relevance of the produced tests (and, by extension, the relevance of the proposed coverage criteria). They pointed out the interest of the graphical visualization of the property automaton which shows, and addresses, various configurations that one might forget when designing tests manually.

Evaluation of fault detection capabilities. For our experiments, we have considered the eCinema model, along with additional properties that aim at illustrating various combinations of scopes and patterns. In addition to properties (1)-(3) presented before, we considered three more specifying that: "before a successful ticket deletion, there eventually exists at least one successful ticket buying" (4), "globally, a successful login precedes a successful logout" (5), and "after having deleted all the tickets, it is not possible to delete one ticket unless buying one in the meantime" (6).

We started by generating a test suite with the Smartesting CertifyIt test generator, and completed this test suite with 11 tests in order to satisfy the different coverage criteria for each properties. We do not detail this process, as it is not the purpose of this paper; however, the interested reader may refer to [10]. We then mutated the original eCinema model, by blindly using the following mutation operators, dedicated to OCL code: Simple Set Operator Replacement (SSOR) replacing a set operator by another, Simple expression Negation Operator (SNO) negating an atomic condition inside a predicate, Stuck-At-False (SAF) replacing predicates by false, and Action Deletion (AD) deleting a predicate of the postcondition (e.g. to remove a variable assignment).

We then run our test suite on these mutant models. We compared, at each step of each test, the expected result of each operation (given by the test case) with the actual returned values (this is how conformance is established on a concrete implementation), giving us a conformance verdict. In addition, we monitored the execution of each test on the automata associated to the considered test properties, to check if the observation was too weak to detect an actual property violation. Finally, we compared our test suite with the original test suite computed by CertifyIt. Figure 6 show the results we obtained. For each considered mutation operator we provide the overall number of mutants that were detected as: conform and not reaching an error state on the automaton (C-NE – it is either an equivalent mutant or

Test suites	Property-Based Testing				Smartesting CertifyIt			
Mutations / Verdicts	C-NE	NC-NA	NC-E	C-A	C-NE	NC-NA	NC-E	C-A
SSOR	2	1	1	2	4	1	1	
SNO		28	2			28		2
SAF		31	1			31	1	
AD	6	12		4	15	8		

Figure 6: Results of the mutant detection for the two considered test suites

a mutant that violates the property but it was not possible to observe it), non-conform but not reaching an error state (NC-NA – it is the cases of mutants that are not related to the property or, at least, not killed for that reason), non-conform and reaching an error state (NC-E – actual violations of the property detected using basic observations), conform but reaching an error state (C-A – actual violations of the property that have not been detected using basic observations, but could be detected by more intrusive means such as monitoring).

Experimental results show that our technique is able to: (1) build test cases that consist in operations leading to a violation of the property (see lines SSOR and AD in which respectively 2 and 4 tests detect the mutant using the property automaton), (2) build test cases that make property violations observable (see line SNO), and (3) build new test cases that are likely to reveal other non-conformances (even if they are not related to the property), improving the overall efficiency of the test suite (see line AD).

7 Related Work

The notion of property-based testing is often employed in the test generation context. Several approaches [15, 21, 1] deal with LTL formulae, that are negated and then given to a model-checker that produces traces leading to a counter-example of this property, and thus defining the test sequences. Our work improves these approaches by defining both nominal and robustness test cases, aiming either at illustrating the property or checking the system's robustness w.r.t. it. A recent work [14] defines the notion of property relevant test cases, introducing new coverage criteria that can be used to determine positive and negative test cases. Nevertheless, our approach proposes several differences. First, we do not rely on LTL, but on a dedicated language easier to manipulate than LTL by non-specialists. Second, the notion of property-relevance is defined at the LTL level, whereas we rely on the underlying automata. Finally, the relevance notion acts as an overlay to classical coverage criteria, while we propose new ones.

Based on Dwyer's work, jPost [13] uses a property expressed in a trace logic for monitoring an implementation. Similarly, in [19] the authors introduce the notion of observers, as ioSTS, that decide the satisfaction of the property and guide the test generation within the STG tool. Our work differs in the sense that the coverage criteria are not only used as monitors for passive testing, but they can also be employed for active testing.

A lot of scenario-based testing works focus on extracting scenarios from UML diagrams, such as the SCENTOR approach [24] or SCENT [20] using statecharts. The SOOFT approach [22] proposes an object oriented framework for performing scenario-based testing. In [3], Binder proposes the notion of round-trip scenario test that cover all event-response path of an UML sequence diagram. Nevertheless, the scenarios have to be completely described. Our approach proposes to automatically generate the test scenarios from higher level descriptions of the properties the validation engineer wants to test. In [2], the authors propose an approach for the automated scenario generation from environment models for testing of real-time reactive systems. The behavior of the system is defined as a set of events. The process relies on an attributed event grammar (AEG) that specifies possible event traces. Even if the targeted applications are different, the AEG can be seen as a generalization of regular expressions. Our approach goes further as it uses a property description language that is close to a natural language.

8 Conclusion and Future Works

In this paper, we have presented a model-based testing process based on test properties. These latter are expressed in a dedicated formalism that captures the dynamics of the system. Each property is translated into an automaton, for which new coverage criteria have been introduced, in order to illustrate the property. In addition, we propose to refine the automaton so as to exhibit specific transitions that are closely related to error traces that are not accepted by the property. This technique makes it possible to introduce a notion of robustness testing to ensure that the property is correctly implemented. We have tool-supported this approach to apply it to UML/OCL models [10]. The advantages of this approach are twofold. Mainly, it provides a means to produce test cases that can be directly related to the property. Such a traceability makes it a suitable approach for industrial purposes. In addition, the automata and their refinements can be used to measure the coverage of corner cases of a property for an existing test suite. This approach has been evaluated in the context of industrial projects, which gave us a very positive feedback on the usefulness of the coverage criteria, exhibiting specific sequences of operations one may want to consider when testing. Finally, notice that the proposed coverage criteria are not specific to UML/OCL and could be adapted to any other notation that would use the same notions of scope and patterns with a different representation of events.

For the future, we first plan to extend our property language to introduce local variables. Such an extension would greatly improve the expressiveness of our property language. However, this extension would imply the definition of data coverage criteria dedicated to the coverage of the properties of these values. Second, we are also investigating a way to efficiently generate the tests satisfying the coverage criteria we proposed. For now, a solution using test scenarios has been implemented. However, the combinatorial unfolding of the scenarios compromises the full automation of the test generation approach.

References

[1] P. Amman, W. Ding & D. Xu (2001): *Using a Model Checker to Test Safety Properties*. In: *7th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'01)*, IEEE, pp. 212–221, doi:10.1109/ICECCS.2001.930180.

[2] M. Auguston, J.B. Michael & M.-T. Shing (2005): *Environment behavior models for scenario generation and testing automation*. In: *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, ACM, New York, NY, USA, pp. 1–6, doi:10.1145/1083274.1083284.

[3] Robert V. Binder (1999): *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet & M. Utting (2007): *A subset of precise UML for model-based testing*. In: *A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing*, ACM Press, London, United Kingdom, pp. 95–104, doi:10.1145/1291535.1291545.

[5] K. Cabrera Castillos, F. Dadeau & J. Julliand (2011): *Scenario-Based Testing from UML/OCL Behavioral Models – Application to POSIX Compliance*. *STTT, International Journal on Software Tools for Technology Transfer* 13(5), pp. 431–448, doi:10.1007/s10009-011-0189-7.

[6] K. Cabrera Castillos, F. Dadeau, J. Julliand, B. Kanso & S. Taha (2013): *A Compositional Automata-based Semantics for Property Patterns*. In E.B. Johnsen & L. Petre, editors: *iFM'2013, 10th Int. Conf. on integrated Formal Methods*, *LNCS* 7940, Springer, Turku, Finland, pp. 316–330, doi:10.1007/978-3-642-38613-8_22.

[7] K. Cabrera Castillos, F. Dadeau, J. Julliand & S. Taha (2011): *Measuring Test Properties Coverage for evaluating UML/OCL Model-Based Tests*. In B. Wolff & F. Zaidi, editors: *ICTSS'11, 23-th IFIP Int. Conf. on Testing Software and Systems*, *LNCS* 7019, Springer, Paris, France, pp. 32–47, doi:10.1007/978-3-642-24580-0_4.

[8] *Smartesting CertifyIt test generator*. `http://www.smartesting.com/`.

[9] D. Clarke, T. Jéron, V. Rusu & E. Zinovieva (2001): *STG: a tool for generating symbolic test programs and oracles from operational specifications*. In: *ESEC/FSE-9: Proc. of the 8th European Software Engineering Conference*, ACM, New York, NY, USA, pp. 301–302, doi:10.1145/503209.503252.

[10] F. Dadeau, K. Cabrera Castillos, Y. Ledru, T. Triki, G. Vega, J. Botella & S. Taha (2013): *Test Generation and Evaluation from High-Level Properties for Common Criteria Evaluations - The TASCCC Testing Tool*. In B. Baudry & A. Orso, editors: *ICST 2013, 6th Int. Conf. on Software Testing, Verification and Validation, Testing Tool track*, IEEE Computer Society Press, pp. 431–438, doi:10.1109/ICST.2013.60.

[11] Richard A. DeMillo (1989): *Completely Validated Software: Test Adequacy and Program Mutation*. In: *ICSE*, pp. 355–356, doi:10.1145/74587.74634.

[12] M. B. Dwyer, G. S. Avrunin & J. C. Corbett (1999): *Patterns in property specifications for finite-state verification*. In: *ICSE'99: Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 411–420, doi:10.1145/302405.302672.

[13] Y. Falcone, L. Mounier, J.-C. Fernandez & J.-L. Richier (2008): *j-POST: a Java Toolchain for Property-Oriented Software Testing*. *Electr. Notes Theor. Comput. Sci.* 220(1), pp. 29–41, doi:10.1016/j.entcs.2008.11.004.

[14] G. Fraser & F. Wotawa (2008): *Using Model-Checkers to Generate and Analyze Property Relevant Test-Cases*. *Software Quality Journal* 16, pp. 161–183, doi:10.1007/s11219-007-9031-6.

[15] A. Gargantini & C Heitmeyer (1999): *Using Model Checking to Generate Tests from Requirements Specifications*. In: *Procs of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pp. 146–162, doi:10.1145/318774.318939.

[16] J. C. Huang (1975): *An Approach to Program Testing*. *ACM Comput. Surv.* 7(3), pp. 113–128, doi:10.1145/356651.356652.

[17] Antti Huima (2007): *Implementing Conformiq Qtronic*. In A. Petrenko, M. Veanes, J. Tretmans & W. Grieskamp, editors: *Testing of Software and Communicating Systems*, *Lecture Notes in Computer Science* 4581, Springer Berlin / Heidelberg, pp. 1–12, doi:10.1007/978-3-540-73066-8_1.

[18] P.-A. Masson & et al. (2010): *An Access Control Model Based Testing Approach for Smart Card Applications: Results of the POSÉ Project*. *Journal of Information Assurance and Security* 5, pp. 335–351.

[19] V. Rusu, H. Marchand & T. Jéron (2005): *Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems*. In J. Fitzgerald, A. Tarlecki & I. Hayes, editors: *Formal Methods 2005 (FM05)*, LNCS, Springer, pp. 189–204, doi:10.1007/11526841_14.

[20] J. Ryser & M. Glinz (1999): *A Practical Approach to Validating and Testing Software Systems Using Scenarios*. Available at `citeseer.ist.psu.edu/ryser99practical.html`.

[21] L. Tan, O. Sokolsky & I. Lee (2004): *Specification-based Testing with Linear Temporal Logic*. In: *IRI'2004, IEEE Int. Conf. on Information Reuse and Integration*, pp. 413–498, doi:10.1109/IRI.2004.1431509.

[22] W. T. Tsai, A. Saimi, L. Yu & R. Paul (2003): *Scenario-based Object-Oriented Testing Framework*. In: *Int. Conf. on Quality Software*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 410–417, doi:10.1109/QSIC.2003.1319129.

[23] J. Warmer & A. Kleppe (2003): *The Object Constraint Language Second Edition: Getting Your Models Ready for MDA*. Addison-Wesley.

[24] J. Wittevrongel & F. Maurer (2001): *SCENTOR: Scenario-Based Testing of E-Business Applications*. In: *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, IEEE Computer Society, Washington, DC, USA, pp. 41–46, doi:10.1109/ENABL.2001.953386.