Measuring Test Properties Coverage for evaluating UML/OCL Model-Based Tests

Kalou Cabrera Castillos, Frédéric Dadeau, Jacques Julliand, Safouan Taha

Measuring Test Properties Coverage for evaluating UML/OCL Model-Based Tests

Kalou Cabrera Castillos[1], Frédéric Dadeau[1], Jacques Julliand[1], Safouan Taha[2]

[1] LIFC / INRIA CASSIS Project – 16 route de Gray - 25030 Besançon cedex, France
Email: kalou.cabrera,frederic.dadeau,jacques.julliand@lifc.univ-fcomte.fr
[2] SUPELEC Systems Sciences (E3S) 3 rue Joliot-Curie, 91192 Gif-sur-Yvette, France
Email: safouan.taha@supelec.fr

Abstract. We propose in the paper a test property specification language, dedicated to UML/OCL models. This language is intended to express temporal properties on the executions of the system, that one wants to test. It is based on patterns, specifying the behaviours one wants to exhibit/avoid, and scopes, defining the piece of execution trace on which a given pattern applies. Each property is a combination of a scope and a pattern, providing a means for a validation engineer to easily express temporal properties on a system, without using complex formal notations. Properties have the semantics of an event-based transition system whose coverage can be measured so as to evaluate the relevance of a given test suite. These principles aim at being used in the context of a research project, in which the security properties are expressed on an industrial case study of a smart card operating system. This approach makes it possible to assist the Common Criteria evaluation of the testing phase, that requires evidences of the extensiveness of the testing phase of a security product.
Keywords: Model-Based Testing, UML/OCL, temporal property, coverage, model animation.

1 Introduction and Motivations

Critical software validation is a challenge in software engineering and a convenient context for Model-Based Testing [4]. Indeed, the cost of writing a formal model to support the test case generation phase is made profitable by the necessity of increasing the confidence in the safety and the security of the system. MBT is well-suited to conformance testing, as the model describes the expected behavior of a system. The system under test is then checked against the model on specific execution traces called test cases. A conformance relationship, usually based on the observation points provided by the SUT, is then used to establish the test verdict.

This work is done in the context of the ANR TASCCC project[3], we are interested in the validation of smart card products security by means of model based

[3] Funded by the French National Research Agency ANR-09-SEGI-014 – http://lifc.univ-fcomte.fr/TASCCC

tests. The tests are produced by the CertifyIt tool, provided by the Smartesting company[4]. This test generator takes as input model based tests in UML/OCL and generated tests aiming at the structural coverage of the OCL code describing the behaviors of the class operations. CertifyIt is an automated test generator, in the sense that, apart from the model, no further information is required to generate the tests.

We propose to consider user-defined test properties to express test patterns associated to the requirements of the software. The contribution is twofold. First, we propose a test property language based on Dwyer's property patterns [9] and applied to UML/OCL models. In this context, the considered events are either controllable (the invocation of an operation) or observable (a state predicate becomes satisfied at a given state). The properties describe the apparition of these events, in given scopes. Second, we propose to assist the validation engineer, by measuring the coverage of the property. Indeed, a property exhibits a set of model executions that are authorized, expressed as an automaton. In order to evaluate the exhaustiveness of the testing phase, we measure and report the coverage of the underlying automaton. In addition, uncovered parts of the property indicate which part has not been tested and, as a consequence, on which part the tester should focus his efforts.

The remainder of the paper is organized as follows. Section 2 presents the test generation process of the CertifyIt tool based on the structural coverage of the OCL code. Then, the property language is defined in Sec. 3 and its semantics is provided in Sec. 4. Section 5 defines the notion of property coverage and explains how this action is processed. Finally, we conclude and present the related and future works in Sec. 6.

2 Test Generation from UML/OCL Models

We present here the test generation principles of the CertifyIt test generation tool. First, we introduce the subset of UML/OCL that is considered and we illustrate it with a simple running example. Then, we present the test generation strategy of CertifyIt.

2.1 Considered Subset of UML/OCL

The model aims at being used by the CertifyIt tool, commercialized by the Smartesting company. This tool generates automatically model-based tests from a UML model [5] 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, 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.

[4] `www.smartesting.com`

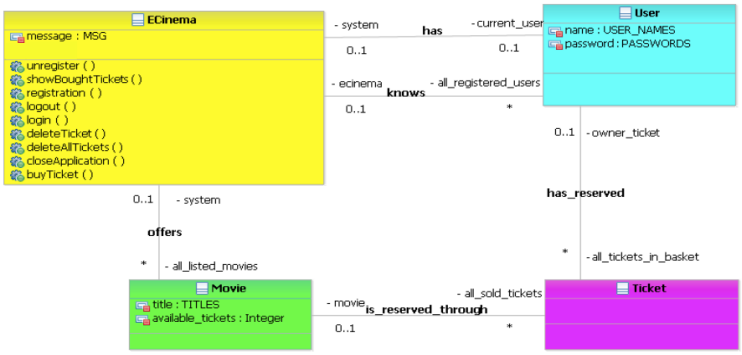

Fig. 1. Class Diagram of the eCinema Model

Concerning modelling, some restrictions apply on the class diagram model and OCL constraints that can be written. The system under test (SUT) has to be modelled by a class, which carries all the operations representing the API provided by the SUT. CertifyIt does not allow inheritance, nor stereotypes like *abstract* or *interface* on the classes. Objects can not be created when executing the model. As a consequence, the object diagram, representing the initial state, has to provide all the possible class instances, possibly isolated (i.e., not associated to the SUT object or any other object) if they are not supposed to exist at the initial state.

OCL provides the ability to navigate the model, select collections of objects and manipulate them with universal/existential quantifiers to build boolean 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. The overall objective is to dispose of an executable UML/OCL model. Indeed, the test cases are produced by animating the model in order to satisfy a given coverage criterion. Before describing this process, we first introduce a simple running example.

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. 1 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. Several requirements are associated to the system, for example: *the user must be registered and connected to access the proposed services*, *the registration is valid*

only if the user's name and password are valid and if the user is not already registered, the user must be connected in order to buy tickets, etc.

The initial state contains a single instance of the system under test class eCinema, identified by `sut`, two instances of movies linked with the `sut` instance (instanciating the `offers` association), twenty-five isolated instances of tickets, and two users, one registered (i.e. linked to the SUT using the `knows` association) and one not registered (isolated instance).

This model contains several operations whose meaning are straightforward: `unregister`, `showBoughtTickets`, `registration`, `logout`, `login`, `deleteTicket`, `deleteAllTickets`, `closeApplication`, `buyTicket`.

Figure 2 shows the OCL code of the `buyTicket` operation used by an authenticated user to buy a ticket. This operation can only be invoked with a valid movie title and if all the tickets are not already assigned to users. To be successful, the user has to be authenticated, and at least one tickets for the movie should be available. Otherwise, error messages report the cause of the failure. Upon successful execution, an available ticket is chosen and assigned to the user for the corresponding movie, and the number of available seats is decremented.

In the OCL code, there are non-OCL annotations, inserted as comments, such as `---@AIM: id` and `---@REQ: id`. The `---@AIM:id` tags denotes test targets while the `---@REQ: id` tags mark requirements from the informal specifications. These tags are used by CertifyIt to know which tags were covered during the execution of the model, and, consequently, inside the test cases.

2.3 CertifyIt Test Selection Criterion

Smartesting CertifyIt is a functional test generator that aims at exercising the atomic transitions of the model, provided by the class operations. The CertifyIt

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

Fig. 2. OCL code of the `buyTicket` operation

test generation strategy works in two steps. First, it identifies test targets and, second, it builds a test case that is a trace, composed of successive operation calls, reaching the considered target.

In the first part of the test generation process, the test targets are computed by applying a structural test selection criterion on the OCL code of the class operations (Decision Coverage criterion). Each test target is thus associated to a predicate that describes the set of possible concrete states from which the operation can be invoked. Each test aims at covering a specific target (possibly characterized by a set of `@AIM` tags). There are three test targets for the `buyTicket` operation, one for each available `@AIM` tags.

The second part of the test generation process consists in performing an automated state exploration, from the initial state, in order to reach a state satisfying the state predicate associated to the test target. This sequence is called a *preamble*. The preamble computation is based on a Breadth First Search algorithms that stops when the targeted state is reached. To obtain the test case, the preamble is concatenated with the invocation of the targeted operation with the appropriate parameter values.

The test selection criterion ensures that one test is built for each target. Nevertheless, some targets may be covered several times, if they are found in the preambles of others targets. Table 1 shows the tests generated for the operation `buyTicket` displayed in Fig. 1.

Notice that the undetermined state of the test $BTTest_3$ declared by CertifyIt refers to internal limitations of the tool in terms of depth search bound. Indeed, in our initial state, we specified that the two movies could deliver 20 tickets each, which needs to build a test sequence in which all 20 tickets are already bought. Since this configuration could be reached (the number of ticket instances is set to 25), this message does not conclude on the general unreachability of the test targets.

Since CertifyIt is a functional test generator, it is not intended to cover specific sequences of operations, or states. Nevertheless, the tool provides a means to complete automatically generated tests with test scenarios, built using the simulator (the internal model animator) and exported as test cases.

3 Test Property Language

The Object Constraint Language is quite similar to first-order predicate logic. OCL expressions are used in invariants, pre- and postconditions. They describe

Test name	Test sequence	Target
$BTTest_1$	`init ; sut.login(REGISTERED_USER,REGISTERED_PWD) ; sut.buyTicket(TITLE1) ;`	`@AIM: BUY_Success`
$BTTest_2$	`init ; sut.buyTicket(TITLE1) ;`	`@AIM: BUY_Login_Mandatory`
$BTTest_3$	`declared as ''Undetermined''`	`@AIM: BUY_Sold_out`

Table 1. Generated tests for the `buyTicket` operation

a single system state or a one-step transition from a previous state to a new state upon the call of some operation.

Several OCL extensions already exist to support temporal constraints [8,11,16]. They only add to OCL unary and binary temporal operators (e.g., always, next and until) in order to specify safety and liveness properties. Unfortunately, most developers are not familiar with temporal logics and this is a serious obstacle to the adoption of such OCL extensions. We propose to fill in this gap by adding to OCL a pattern-based temporal layer to ease the specification of temporal properties.

3.1 A temporal extension to UML/OCL

For specifying the temporal aspects of system properties, we adopt the work of Dwyer et al. [9] on specification patterns for temporal properties. Although formal methods are largely automated today, most engineers are not familiar with formal languages such as linear temporal logic (e.g. LTL) or tree logic (e.g. CTL). The effort required to acquire a sufficient level of expertise in writing these specifications represents a serious obstacle to the adoption of formal methods. Therefore, Dwyer et al. have introduced a new property specification language based on patterns in which a temporal property is a combination of one pattern and one scope.

Patterns There are 8 patterns organized under a semantic classification. We distinguish occurrence patterns from order patterns.

Occurrence patterns are: (*i*) *Absence*: an event never occurs, (*ii*) *Existence*: an event occurs at least once, (*iii*) *Bounded Existence* has 3 variants: an event occurs k times, at least k times or at most k times, and (*iv*) *Universality*: an event/state is permanent.

Order patterns are: (*v*) *Precedence*: an event P is always preceded by an event Q, (*vi*) *Response*: an event P is always followed by an event Q, (*vii*) *Chain Precedence*: a sequence of events $P_1, \ldots, P_n$ is always preceded by a sequence $Q_1, \ldots, Q_m$ (it is a generalization of the Precedence pattern), (*viii*) *Chain Response*: a sequence of events $P_1, \ldots, P_n$ is always followed by a sequence $Q_1, \ldots, Q_m$ (it is a generalization of the Response pattern).

Scopes A scope is the discrete time interval over which the property holds. There are five kinds of scopes, illustrated on Fig.3 (taken from [9]):

(*a*) `globally` covers the entire execution, (*b*) `before` Q covers the system's execution up to the first occurrence of Q, (*c*) `after` Q covers the system's execution after the first occurrence of Q, (*d*) `between` Q `and` R covers time intervals of the system's execution from an occurrence of Q to the next occurrence of R, (*e*) `after` Q `until` R is the same as the `between` scope in which R may not occur. Dwyer et al. provide a complete library [5] mapping each pattern/scope combination to the corresponding formula in many formalisms (LTL, CTL, μ-calculus,

[5] `http://patterns.projects.cis.ksu.edu`

TempExpr ::= *TempPattern* *TempScope*

TempPattern ::= `always` OclExpression
| `never` *Event*
| `eventually` *Event* (*Times*)?
| *Event* (`directly`)? `precedes` *Event*
| *Event* (`directly`)? `follows` *Event*

TempScope ::= `globally`
| `before` *Event*
| `after` *Event*
| `between` *Event* `and` *Event*
| `after` *Event* `until` *Event*

Event ::= *ChangeEvent* (‖ *Event*)?
| *CallEvent* (‖ *Event*)?

ChangeEvent ::= `becomesTrue(`OclExpression`)`

CallEvent ::= `isCalled(` (name`::`)? name
(`, pre:` OclExpression)?
(`, post:` OclExpression)?
(, TagList)?)

TagList ::= `including:` { Tags}
| `excluding:` { Tags}

Times ::= integer `times`
| `at least` integer `times`
| `at most` integer `times`

Tags ::= `@REQ:` name (`,` *Tags*)?
| `@AIM:` name (`,` *Tags*)?

Fig. 4. Syntax of our temporal property extension

etc.). For example, one entry of this library that maps the *Response* pattern S `follows` P to LTL formula for different scopes is given in Tab. 2.

The work of Dwyer et al. on such patterns dramatically simplifies the specification of temporal properties, with a fairly complete coverage. Indeed, they collected hundreds of specifications and they observed that 92% fall within this small set of patterns/scopes [9]. For these reasons, we adopt this pattern-based approach for the temporal part of our OCL extension. We now present its syntax.

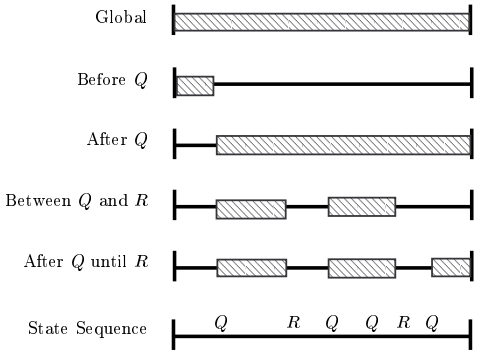

Fig. 3. Property scopes

3.2 Syntax

We extended the OCL concrete grammar defined within the OMG standard [15] in order to express temporal properties that will provide our test properties.

The syntax of our language is summarized in Fig. 4. In this figure, non-terminals are designated in *italics*, terminals are underlined and construct (...)?

Scope	LTL
`globally`	$\Box(P \Rightarrow \Diamond S)$
`before` R	$\Diamond R \Rightarrow (P \Rightarrow (\neg R \,\mathsf{U}\, (S \wedge \neg R))) \,\mathsf{U}\, R$
`after` Q	$\Box(Q \Rightarrow \Box(P \Rightarrow \Diamond S))$
`between` Q `and` R	$\Box((Q \wedge \neg R \wedge \Diamond R) \Rightarrow (P \Rightarrow (\neg R \,\mathsf{U}\, (S \wedge \neg R))) \,\mathsf{U}\, R)$
`after` Q `until` R	$\Box(Q \wedge \neg R \Rightarrow ((P \Rightarrow (\neg R \,\mathsf{U}\, (S \wedge \neg R))) \,\mathsf{W}\, R)$

Table 2. LTL mapping of the S `follows` P pattern

designates an optional part. Terminal name designates an identifier that represents the name of a temporal property, an instance name, an operation name (when separated by `::`) or tag names (`REQ` or `AIM`, as shown in Sec. 2.3). The OCLExpression terminal designates an OCL predicate according to the supported syntax of UML4ST (as explained in Sec. 2.1).

As explained before, a temporal property *TempExpr* is a combination of a pattern *TempPattern* and a scope *TempScope* whose respective meanings have been informally described before. Events are of two kinds. A *ChangeEvent* is parameterized by an OCL predicate P, and designates a step in which P now becomes true, i.e. P was evaluated to false in the preceding step. This event represents an observable event, that is triggered after the execution of an operation of the system (but it is not possible to know *a priori* which operation will cause this event). A *CallEvent* represents the invocation of an operation on a given instance. Optional field `pre` provides a precondition, namely an OCL predicate that has to be true before the invocation of the operation (or at the beginning of the invocation if the expression refers to input parameter values). Optional field `post` provides a postcondition that has to be true after the execution of the operation. Finally, optional field `including` (resp. `excluding`) provides the set of tags for which at least one has to be covered (resp. none shall be covered) by the execution of the operation. For example, event `isCalled(sut::buyTicket, pre: in_title=TITLES::Tron and not self.current_user.oclIsUndefined(),including:{@AIM:BUY_Success})` is triggered when operation `buyTicket` is invoked on the `sut` instance, with parameter `in_title` representing a given movie title (provided as an enumeration class `TITLES`), when a user is logged on the system and the operation terminates by a successful buying of a ticket for this movie.

Example 1 (Temporal property). Let us consider the example of the eCinema application described in Sec. 2.2. We can formalize the following test requirements of the application as temporal properties. *Users can only buy tickets when logged on the system.* This statement can be expressed as a test property using a `between` scope as follows:

```
eventually isCalled(buyTicket, including:{@AIM:BUY_Success})
           at least 0 times
   between becomesTrue(not(self.current_user.isOclUndefined()))
   and becomesTrue(self.current_user.isOclUndefined()).
```

Even though the presence of `at least 0` in the pattern may be strange, it describes the optional occurrence of the event C. But, this statement may also be expressed as a robustness test property:

```
never isCalled(buyTicket, including:{@AIM:BUY_Success})
   after becomesTrue(self.current_user.isOclUndefined())
   until isCalled(login, including:{@AIM:LOGIN_Success}).
```

4 Semantics of the Test Property Language

In this section, we formally define *substitution automata* that are used to describe the semantics of the properties, and we give a definition of the substitution process. The motivation behind the use of such automata is to have a compositional semantics, so as to be able to build an automaton representing a test property. The latter is thus a combination of a scope and a pattern. Each of them is formalized using substitution automata defined in Def. 1. The resulting automaton will capture all the executions of the system and highlight specific transitions representing the events used in the property. We first define the substitution automata modelling scopes and patterns. Then we present the labels of the transitions representing the events in properties. Finally, we give the definition of the substitution that applies.

4.1 Substitution automata

Substitution automata are labelled automata where the labels are defined from the events (see *Event* in Fig. 4). The states in S are substitution states that represent a property provided with generic patterns. They will be replaced by an automaton defining a particular pattern. For some property, such as $Pattern$ `between` E_1 `and` E_2, it is necessary to avoid that E_2 is triggered in the automaton of the pattern. For that, we formalise this restriction (R) labelling the substitution state by a set of labels. These labels are events that will not be triggered by the internal transitions of the pattern automaton.

Definition 1 (Substitution automaton). *Let Σ be the set of labels. A substitution automaton is a 6-tuple $a = \langle Q, F, q_0, S, R, T\rangle$ where: Q is a finite set of states, F is a set of final states ($F \subseteq Q$), q_0 is an initial states ($q_0 \in Q$), S is a set of substitution states ($S \subseteq Q$), R is a function that associates a set of labels to any substitution state ($R \in S \rightarrow \mathcal{P}(\Sigma)$), T is a set of transitions ($T \in Q \times \mathcal{P}(\Sigma) \times Q$) labelled by a set of labels.*

Graphically, substitution states will be depicted as squares (instead of regular circles). If the substitution state presents a restriction on the alphabet, the restricted elements are written as state labels.

Scopes and patterns will be formalized as substitution automata whose labels are events that occur in the scopes and patterns. More precisely, a scope is modelled as a substitution automaton with one substitution state, whereas a pattern is modelled as a substitution automaton without substitution states.

Example 2 (Substitution Automaton for Scopes and Patterns). Figures 5 and 6 respectively give the graphical representation of the automata of the temporal property TP `between` A `and` B in which the square state represents the generic pattern TP and pattern `eventually` C `at least` 0 `times`. For the latter, the automaton clearly identifies a reflexive transition that represents the occurrence of C. Its presence originates from the motivation of coverage measure, and will make it possible to see if C has been called, or not.

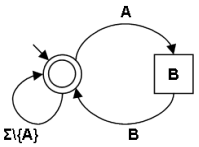

Fig. 5. `between` A `and` B

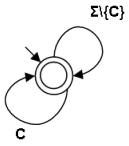

Fig. 6. `eventually` C `at least` 0 `times`

Before describing the substitution operation, we now present the alphabet of labels that is used in our automata.

4.2 Events and Labels

The events triggered in a property description and described by the rules *CallEvent* and *ChangeEvent* in the grammar described in Fig. 4 are represented by labels in automata.

The *isCalled* structure can be divided in four parts: the operation name, the precondition for this call to happen, the postcondition that has to be satisfied after the call, and a list of tags that may be activated. Thus, we will use the following notation to denote the alphabet elements in our automata:

$$[operation\ name, precondition, postcondition, tags].$$

Note that there can be unspecified slots in this notation: for example, one can specify the activation of a specific operation *op* provided with a precondition *pre*, but leaves the postcondition and tags slots free, denoted by [*op*,*pre*,_,_], in which symbol "_" designate a free slot, meaning that it can be any operation, the predicate `true` and any tag. A such label represent a set of atomic events that are triggered by the transition. For example, in example 3 the label C represent one atomic event whereas the label A represent the set of atomic events for any tag in any operation. Also, the tag list references the tags that may be activated. If the *isCalled* specifies a list of excluded tags, we can extract the set of tags that may be used (the complementary set of tags) as we know all the tags of a specific operation from the model.

The *ChangeEvent* can be seen as a free operation call with a pre- and postcondition without specifying the operation name nor the tags to be used. Notice that any operation can be called. The precondition is the negation of the `becomesTrue` predicate, illustrating the fact that, before the call, the execution of the operation has made the predicate become true. Therefore, all `becomesTrue(`P`)` events can be denoted by the label [_,$\neg P$,P,_].

Each scope and pattern are associated to a skeleton of substitution automata whose transition labels are instantiated with the events appearing in the property that they define.

Example 3 (Labels). Consider the first property given in Example 1. According to the label notation introduced in Sec. 4.2, the labels A, B and C in Fig. 5 and Fig. 6 are respectively [_, `self.current_user.isOclUndefined()`, `not(self.current_user.isOclUndefined())`, _], [_, `not(self.current_user.isOclUndefined())`, `self.current_user.isOclUndefined()`, _] and [`buyTicket`, _, _, {`@AIM:BUY_Success`}].

Notice that two labels can be the same set of events even if they are not written with the same 4-uplet. It is also possible that the set of atomic events of a label includes the set of atomic events of another label. In practice, these cases must represent an ambiguous test property. Therefore, we assume that all the events labelling the outgoing transitions of the automata states are mutually exclusive, producing, in that, deterministic automata.

When the two automata, the scope and the pattern, have been defined, they have to be combined to obtain a single automaton, that does not contain any substitution state. We now describe how substitution works.

4.3 Substitution

The substitution operation defined in Def. 2 replaces the substitution state s, representing a generic pattern, by an automaton as, representing an instance of a pattern, in an automaton a, representing a test property. For an automaton a, we denote by X_a the component X of a. For the definition, we assume that there is only one substitution state in a and no substitution state in as, i.e. $S_{as} = \emptyset$ and $R_{as} = \emptyset$. We also assume that the label of transitions in a and in as are different and that the set of states of a and as are disjoint.

The set of states of the resulting automaton c is the set of states of a without its substitution states S_a union each state of the substituted automaton as. When s is a final (resp. initial) state, the set of final (resp. initial) states of c is the set of a without its final (resp. initial) substitution states union each final (resp. initial) state of the substituted automaton as. Otherwise, the set of final (resp. initial) states is this of a. c contain no substitution state and consequently no restriction.

We denote by q a non-substitution state ($q \in Q - S$), s a substitution state ($s \in S$) and E a set of labels, the transitions of c are defined in four cases:

1. any transition $q \xrightarrow{E} q'$ in a is a transition of c,
2. for a transition $q \xrightarrow{E} s$ in a, there is a transition $q \xrightarrow{E} q'$ in c for the initial state of as,
3. for a transition $s \xrightarrow{E} q'$ in a, there is a transition $q \xrightarrow{E} q'$ in c for any final state q of as,
4. any transition $q \xrightarrow{E'} q'$ in as becomes a transition $q \xrightarrow{E} q'$ in c where E is the set of labels E' reduced by the labels $R(s)$.

Definition 2 (Substitution Operation). *Let a be an automaton such that $S_a = \{s\}$. Let as be an automaton such that $S_{as} = \emptyset$. The substitution of the state s by the automaton as in a is the automaton c defined as:*

- $Q_c = (Q_a - S_a) \cup Q_{as}$,
- *when* $s \in F_a$, $F_c = (F_a - \{s\}) \cup F_{as}$, *otherwise* $F_c = F_a$,
- *when* $s = q_{0_a}$, $q_{0_c} = q_{0_{as}}$, *otherwise* $q_{0_c} = q_{0_a}$,
- $S_c = \emptyset$ *and* $R_c = \emptyset$,
- $q \xrightarrow{E} q' \in T_c$ *if and only if:*
  1. $q, q' \in Q_a - S_a$ *and* $q \xrightarrow{E} q' \in T_a$,
  2. $q \in Q_a - S_a$ *and* $q \xrightarrow{E} s \in T_a$ *and* $q' = q_{0_{as}}$,
  3. $q' \in Q_a - S_a$ *and* $s \xrightarrow{E} q' \in T_a$ *and* $q \in F_{as}$,
  4. $\exists E' \in \mathcal{P}(\Sigma)$ *such that* $q \xrightarrow{E'} q' \in T_{as}$ *and* $E = E' - R(s)$.

Example 4 (Substitution Automata Composition). Consider again the scope and the property substitution automata represented in Fig. 5 and Fig. 6. Figure 7 shows the flattened automaton obtained by applying the substitution operation. This is the automaton of the property given in Example 1.

The resulting automaton represents the executions of the system allowed by the property. We now measure the coverage of this automaton to establish a metrics that will be used to evaluate the exhaustiveness of a test suite.

5 Property Coverage Measure

This section presents the technique used to measure the coverage of the property. It is based on the semantics of the property language that was previously introduced.

5.1 Automata Completion

Before performing the coverage measure, we need to complete our property automaton so as to match every possible event on our property automaton. In the example given in Fig. 7, the automaton is complete in the sense that any event will be matched from any state of the automaton. Nevertheless, in practice, the automaton is not necessarily complete. Indeed, the substitution can result in an incomplete automaton: it only represents all valid paths for the property. The complete form of the automaton thus represents all possible paths, including all faulty (with respect to the property) execution. The completion process simply

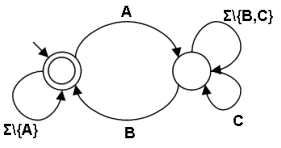

Fig. 7. Graphical representation of the composition for property `eventually` C `at least` 0 `times between` A `and` B

creates a new state that can be seen as a rejection state while final states represent acceptance states. If a state does not allow the triggering of a transition for an alphabet element, we create a transition from this state to the newly created state. Figure 8 illustrates this completion where the rejection state is the circle marked of one cross.

Remark. Ideally, in a Model-Based Testing process, the model describes faithfully the system, and it has no risk of violating a given (temporal) property that is supposed to hold on the system. However, in reality, the model may contain faults and thus, invalidate the property. The completion of the automaton is used to capture events that lead to an invalidate state, which detects the violation of the property. If the model satisfies the property, these additional transitions are useless, since they will never be activated on the model. Nevertheless, we add them to be able to detect possible violations of the property w.r.t. the model execution, indicating a fault in the model or in the property. Thus, we are able to partially verify (i.e. test) the model using existing test sequences, by monitoring the absence of property violation.

5.2 Performing the Measure

The evaluation of the property coverage is based on the coverage of its underlying automaton. Using the CertifyIt animation engine[6], it is possible to replay a set of existing test cases on a model. At each step (i.e. after each operation invocation), the corresponding state can be used to evaluate a given OCL predicate.

The algorithm for measuring the coverage of the property is quite straightforward, and sketched in Fig. 9. This algorithm takes as input a model M, a test suite TS and a completed substitution automaton A, supposed to represent a property. For each test, the automaton exploration starts from its (single) initial state. At each step of the test, the corresponding event on the automaton is matched. If it triggers an outgoing transition from the current state, then the exploration of the automaton progresses, and corresponding transition and states are marked by the test, and the corresponding step. When an rejection state is reached, the exploration stops and an error is returned. Once all the steps have been performed, the algorithm moves to the next test. In the end, we have, for each test, the states and transitions of the automaton reached by the test.

[6] Provided in the context of the TASCCC project by the Smartesting company

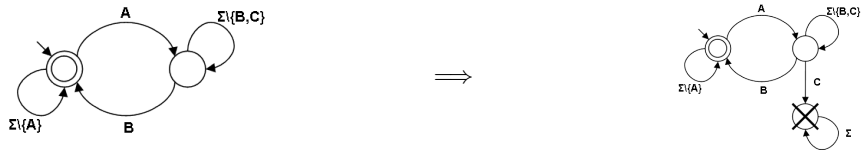

Fig. 8. Completion with rejection states (automaton for never C between A and B)

```
input Model M, Test Suite TS, Automaton a
begin
  for each Test t ∈ TS do
    cover ← false
    currentState ← q0_a
    mark currentState as covered by ⟨t, init⟩
    for each Step st of t do
      tr ← find transition triggered by step st
      next ← tr.destination
      if next is a rejection state then
        throw error("Model does not respect the property");
      elseif next ≠ q0_a then cover ← true
      end if
      mark next state and tr as covered by ⟨t, st⟩
      currentState ← next
    done
    if not cover or currentState ∉ F then remove marking of t end if
  done
end
```

Fig. 9. Coverage Measure Algorithm

When the test is replayed and the coverage of the underlying automaton is performed, three alternatives may happen: (*i*) the test leads to a rejection state, then the model does not respect the property, (*ii*) the test explores at least one transition of the automaton and reaches at least one final state (i.e. it should not stay in a initial and final state), then we say that the test "covers" the property, (*iii*) the test explores the automaton but does not reach any final state (except a final state that is also an initial state), then the test does not cover the property.

Finally, we measure classical automata coverage criteria (all-nodes, all-edges, etc.) and report the coverage of a test suite w.r.t. these criteria. Notice that only the test covering the property are considered.

Example 5 (Measure of the CertifyIt test suite coverage). As explained before, the CertifyIt test generation strategy aims at producing functional test suites. Consider the three test cases dedicated to the `buyTicket` operation, for which we want to evaluate their relevance w.r.t. the two test properties represented by the two automata depicted in Fig. 7 and Fig. 8. None of these tests do cover the property as defined in (*ii*) hereabove, as they never reach a final state (the only final state covered is also initial) in which the user disconnects from the system.

A test suite that satisfies the all-nodes and all-edges coverage criterion for the first property could be the following:

```
{ init; sut.login(REGISTERED_USER,REGISTERED_PWD); sut.showBoughtTickets();
sut.logout(), init; sut.login(REGISTERED_USER,REGISTERED_PWD);
sut.buyTicket(TITLE1); sut.logout(); }
```

6 Conclusion, Related and Future Works

We have presented in this paper a property language for UML/OCL models, based on scopes and patterns, and aiming at expressing test properties. We have proposed to evaluate the relevance of a given test suite by measuring the coverage of an automaton representing the admissible traces of the model's execution that

cover the property. This approach is tool-supported (a tool prototype has been made available for the members of the TASCCC project) and experimented on the GlobalPlatform[7] case study, a last generation smart card operating system, provided by Gemalto[8].

This approach has several interesting features. First, the evaluation of a test suite is relevant w.r.t. a Common Criteria evaluation [7] of a security product which requires specific security requirements to be covered by the test cases during the validation phase. Second, the uncovered parts of the properties that have been discovered indicate precisely on which part of the system the validation engineer has to focus to complete the existing test suite. In addition, the existing tests can be used to identify relevant pieces of model executions that make it possible to reach a given state in the property automaton, helping the validation engineer to design complementary test cases, that do not only rely on its own interpretation of the property.

Related works. This approach is inspired from property monitoring approaches. Many related works fall into the category of passive testing, in which properties are monitored on a system under test [2,1]. This kind of approach is particularly used in security testing, where the violation of security properties can be detected at run-time and strengthen the test verdict [3]. The closest work is reported in [10] which uses a similar approach, also based on Dwyer's property patterns and the classification of occurrence/precedence patterns, in order to monitor test properties. Our approach differs in the sense that we aim at evaluating test cases w.r.t. properties. Also, [14] proposes the generation approach of relevant test sequences from UML statecharts guided by temporal properties. A test relevance criterion is also defined. In [13], temporal properties written in Java Temporal Pattern Language (also inspired by Dwyer's patterns) are designed and translated into JML annotations that are monitored during the execution of Java classes.

Future works. We are currently investigating the way of building these missing test cases automatically using a Scenario-Based Testing approach [6]. In addition, we are also looking for the automated generation of robustness test cases, to be extracted from these user-defined test properties, by using a mutation based testing approach applied to the property, also coupled with a Scenario-Based Testing approach. Another extension of this work will be to define dedicated temporal property coverage criteria e.g. inspired from [12].

References

1. J. A. Arnedo, A. Cavalli, and M. Núñez. Fast testing of critical properties through passive testing. In *Proc. of the 15th IFIP international conference on Testing of communicating systems*, TestCom'03, pages 295–310, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] `www.globalplatform.org/specifications.asp`

[8] `www.gemalto.com`

2. J.M. Ayache, P. Azema, and M. Diaz. Observer: a concept for on-line detection of control errors in concurrent systems. In *9th Sym. on Fault-Tolerant Computing*. 1979.
3. E. Bayse, A. Cavalli, M. Núñez, and F. Zaidi. A passive testing approach based on invariants: application to the wap. In *Computer Networks*, volume 48, pages 247–266. Elsevier Science, 2005.
4. B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.
5. F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing*, pages 95–104, London, UK, July 2007. ACM.
6. K. Cabrera Castillos, F. Dadeau, and J. Julliand. Scenario-based testing from UML/OCL behavioral models – application to POSIX compliance. *STTT, International Journal on Software Tools for Technology Transfer*, 2011. Special Issue on Verified Software: Tools, Theory and Experiments (VSTTE'09). To appear.
7. Common Criteria for Information Technology Security Evaluation, version 3.1. Technical Report CCMB-2009-07-001, july 2009.
8. M. V. Cengarle and A. Knapp. Towards OCL/RT. In *Symposium of Formal Methods Europe*, pages 389–408. Springer, 2002.
9. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE'99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
10. Y. Falcone, J.-C. Fernandez, T. Jéron, H. Marchand, and L. Mounier. More testable properties. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 30–46, Berlin, Heidelberg, 2010. Springer-Verlag.
11. S. Flake and . Mueller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Software and Systems Modeling (SoSyM), Springer*, 2:186, 2003.
12. G. Fraser and F. Wotawa. Complementary criteria for testing temporal logic properties. In C. Dubois, editor, *Tests and Proofs*, volume 5668 of *Lecture Notes in Computer Science*, pages 58–73. Springer Berlin / Heidelberg, 2009.
13. A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java modeling language. *IET Software*, 2(6):500–514, December 2008.
14. S. Li and Z.-C. Qi. Property-oriented testing: An approach to focusing testing efforts on behaviours of interest. In S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, editors, *Testing of Component-Based Systems and Software Quality, Proceedings of SOQUA 2004*, volume 58 of *LNI*, pages 191–206. GI, 2004.
15. Object Management Group. Object Constraint Language. `http://www.omg.org/spec/OCL/2.2`, February 2010.
16. P. Ziemann and M. Gogolla. An OCL Extension for Formulating Temporal Constraints. Technical report, Universität Bremen, 2003.