# Temporal property patterns for model-based testing from UML/OCL

Frédéric Dadeau, Elizabeta Fourneret, Abir Bouchelaghem



# Temporal Property Patterns for Model-Based Testing from UML/OCL[☆]

Frédéric Dadeau[a,*], Elizabeta Fourneret[a,b], Abir Bouchelaghem[a]

[a] *University of Bourgogne Franche-Comté – FEMTO-ST Institute CNRS UMR 6174*
*16 route de Gray, 25030 Besançon cedex, France*
[b] *Smartesting Solutions & Services, 18 rue Alain Savary, 25000 Besançon cedex, France*

**Abstract**

This article describes a new property- and model-based testing approach using UML/OCL models, driven by temporal property patterns and a tool for assisting the temporal properties formalization. The patterns are expressed in the TOCL language, an adaptation of Dwyer's property patterns to OCL. The patterns are used to formalize temporal requirements without having to learn a complex temporal logics such as LTL or CTL. From these properties, automata are automatically computed. These can be used for two purposes. First, it is possible to evaluate the quality of a test suite by measuring the coverage of a property using its associated automaton. Second, the automaton can be used to drive the test generation in order to produce complementary test cases. To this end, we defined dedicated coverage criteria, targeting specific events of the property, and aiming either at illustrating the expected behaviour of the system, or checking its robustness w.r.t. the property. However, it was observed that the semantics of the property language may be more subtle that it seems. To facilitate the adoption of the language by industrials, we have proposed a tool-supported assistant for property design, aiming to help the validation engineer choosing which constructs faithfully correspond to his intention. This approach has been experimented on several case studies with industrial partners. It has shown its interest for software validation, providing useful information thanks to adequate traceability features.

*Keywords:* behavioural model, property patterns, coverage measure, test generation, property design

[☆] This article is an extension of a 2014 paper published at the Model-Based Testing workshop. The extension consists in the experiments section, and the complement on the assistance on properties modelling.

This work has been partially funded by the French National Research Agency (ANR) under grant ANR13-ASMA-0003.

[*] Corresponding author
*Email addresses:* `frederic.dadeau@femto-st.fr` (Frédéric Dadeau), `elizabeta.fourneret@{femto-st.fr, smartesting.com}` (Elizabeta Fourneret)

## 1. Introduction

Model-Based Testing (MBT) [1] consists in using a model to generate test cases, and compute the test verdict, in terms of expected behavior of the system under test (SUT). Models are designed based on the informal requirements of the system, and exploited by model coverage criteria to compute test cases. In addition, the models make it possible to compute the test oracle, namely the expected result of the test. After a concretization step, the abstract tests can then be executed on the SUT and the test verdict can be automatically assigned. MBT is thus a convenient way to automate test generation, and, to some extent, test execution. Various approaches for MBT exist [2], based on different formalisms using, for example, automata (mealy machines, IOLTS, IOSTS), or pre/postconditions notations (B, VDM, JML, UML/OCL), etc. Associated to them, test selection criteria make it possible to generate test cases that guarantee a given level of assurance that the system has been sufficiently exercised.

The work presented in this article aims to improve an existing MBT approach, based on UML/OCL models, that initially targets functional testing. This first approach has been developed and successfully transferred into the industry as the Smartesting CertifyIt test generator. This tool works by automatically applying a structural test selection criterion on the model, namely the branch coverage of the OCL specification [3] of operations contained in a 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. These consist in scenario-based testing approaches that aim to exercise more extensively specific parts of the considered system. Such test scenarios are expressed in a dedicated textual language that describes 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 [4]. 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 the notion of *property patterns* that can be used to express dynamic behaviours of the systems without employing complex temporal logics formalisms [5]. Patterns are intended to provide a means to capture the best practices, or most common practices, of validation engineers designing temporal properties. Patterns are close to natural language and thus, their adoption by non-specialists in formal methods, is expected to be facilitated. They are expressed with a *scope*, that delimits the considered fragments of the execution of the system, and a *pattern* that expresses occurrences, absences or chains of given events inside the scope. These properties are mainly used for verification [5], or passive testing (monitoring) [6]. Our key idea is to use

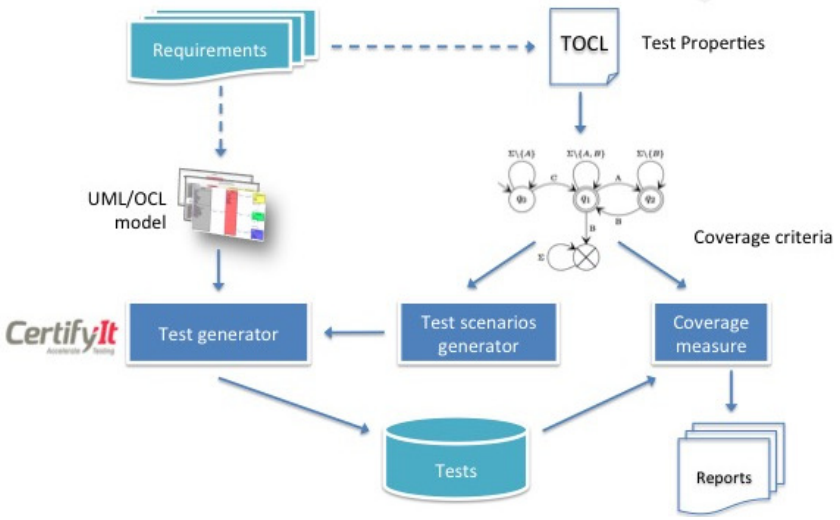



Figure 1: Process of the property-based testing approach

these property patterns for active testing, to drive the test generation. To achieve that, we have designed a temporal property language based on property patterns, named TOCL (for Temporal OCL) [7, 8], aiming to be used with UML/OCL models and the Smartesting CertifyIt test generator.

The proposed process is depicted in Fig. 1. TOCL properties are designed from the initial requirements of the system. These properties are automatically translated into automata, that can be used for two purposes. First, it is possible to evaluate the relevance of a given test repository w.r.t. the property by measuring the property coverage, using its associated automaton. At this stage, a precious feedback can be given to the user to determine how the considered property, and its associated requirements, have been tested. Second, the properties can be used to generate complementary test cases, that target uncovered parts of the property automata.

In this article, we propose three contributions:

- two kinds of dedicated property automata coverage criteria:
  - a first set of criteria that is inspired from classical automata coverage criteria and aims to characterize relevant tests highlighting the behaviours described in the property;
  - a second, mutation-based, coverage criterion that targets corner cases of the property and aims to provoke unexpected events in order to validate the robustness of the system;
- an experimental evaluation of this approach on a realistic case study of the PKCS#11 standard, and
- a tool that assists the validation engineer in writing TOCL properties,

based on a decision tree that drives the choice of the appropriate property patterns.

This article is organised as follows. Section 2 presents the context of the present work, namely the historical approach implemented in the CertifyIt test generator along with the considered subset of UML/OCL and the limitations of this approach. Section 3 presents the TOCL language that we introduced previously. Based on this language, we describe in Section 4 several test selection criteria that target either the nominal coverage of the property, to select tests that illustrate the property, or robustness coverage, to select tests that attempt to violate the property. The three possible usages of these test selection criteria are presented in Section 5. The semantical issues of the language and the subsequent assistant for designing TOCL properties is presented in Section 6. Then, Section 7 describes the experimental assessment that we performed, on a realistic case study of the PKCS#11, and additional experience reports from research projects in which TOCL properties were used. Related work is summarized and compared to our approach in Section 8. Finally, Section 9 concludes and presents the future works.

## 2. Context: Functional Testing from UML/OCL

This section presents the initial functional testing approach proposed by the CertifyIt test generation tool. We first introduce the considered subset of UML/OCL, before presenting the functional test generation process. Finally, we present the functional test generation process that we aim to complement.

### *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 [9] with OCL specifications 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 [3], 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.

Finally, it is important to notice that the semantics of OCL has been modified to consider it as an action language. Thus, the equality operator is interpreted differently depending of the context of the expression in which it appears. If used in a conditional statement (IF), it is evaluated as an equality test. If used outside a conditional statement, it is interpred as an assignement of the left-hand side attribute with the right-hand side value. This allows to write more complex specifications, and improves the expressiveness of the OCL postconditions.

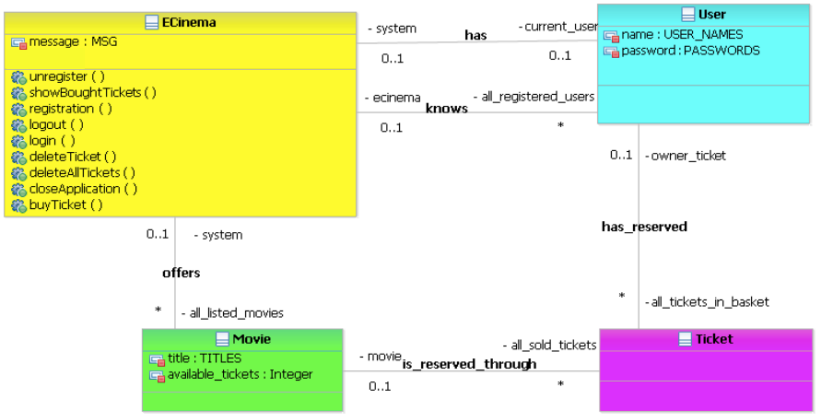



Figure 2: Class diagram of the eCinema model

*UML4ST vs. standard UML.* UML4ST is based on the UML notation (for class diagrams and OCL constraints), but it differs from the usual UML standards. Regarding the OCL semantics, UML4ST does not consider the third logical value *undefined* that is part of the standard OCL semantics. UML4ST relies on existing constructs of OCL but apply a different semantics. Notably, expression `isOclUndefined()` is applicable on 0..1-multiplicity roles sides and is used to check if no object is linked with the current instance. In UML4ST, `null` values can exist when a link to a given object does not exist (multiplicity 0..1 in the class diagram). Comparisons with null values are authorized, but only for objects (contrary to the UML standard, null values for primitive types are not supported). Nevertheless, all expressions have to be defined at evaluation-time in order to be evaluated. The use of a null value in an expression (for example o.x where o is null) will generate an error in the OCL interpreter of the tool when the OCL code is being processed for test generation (or, more generally, when animating the model, which is the underlying technique for test generation). In this case, the tool will request the user to ensure that the accessed object is not null by guarding the access to the objects members with an IF statement or a precondition.

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 the user wants to buy a ticket. The operation first checks that a user is logged on the application, and then checks if there exists an unallocated ticket for this movie. If all these verifications succeed, a ticket is associated to the user. This operation is specified in a defensive style: its precondition is always true, and the postcondition is in charge of distinguishing nominal cases from erroneous cases. We assume in the rest of the article that all operations are specified this way, which is realistic in the context of MBT.

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 particular 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.

### *2.3. Functional Test Generation from UML/OCL with Smartesting CertifyIt*

We present in this section the structural test selection criterion based on behavioral coverage that is implemented within the Smartesting CertifyIt test generator.

The test generator takes as input the MBT model and computes the test targets to cover, by considering structural decision coverage criterion. Thus, each control path of the control flow graph of the operation represents a behavior of the operation. As OCL does not contain any iterative structure, and since we

```
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

do not consider recursive operation calls, the number of paths (only introduced by if-then-else structures) is finite. A control-flow graph is built based on the conjunction of the pre- and postcondition described by the OCL code. Finally, each test target is identified by a set of tags (labelling the considered control path), which refer to a requirement covered by the behavior.

Figure 4 presents the test targets for the `buyTicket` operation described in Fig. 3. Target 1 represents the case when the user is not connected to the reservation system. Target 2 represents a logged user, who attempts to buy a ticket for a movie that is already sold out. Finally, target 3 represents a successful purchase of a ticket, from an authenticated user who requested a ticket for a movie for which places are still available.

From these test targets, the test generation engine will compute a test case, as a sequence of operations that, from the initial state, reaches a state satisfying the guard of the behavior. To achieve that, CertifyIt uses an SMT solver that performs symbolic animation. This technique consists in simulating the execution of the model using symbolic parameters. Each operation activation gathers constraints (the path conditions in the operation code) that are evaluated by the solver to check if there exists an instantiation of the symbolic variables that satisfies these constraints. If a solution is found, the considered sequence of operations can be executed to reach the target state. The test case finally ends with the incovation of the operation from which the test target was extracted.

A test case, such as given in Figure 5, is defined as a sequence of steps. Each step is defined as a tuple $(op, parameters, tags)$ in which $op$ designates the operation that is invoked, $param$ is the instantiation of the parameters, and $tags$ is the set of tags that are covered by this invocation. The test verdict can be established by using the return values of the operations. In addition, the model may contain a specific kind of operations, called observations, that can be used to observe internal model state variables, to be compared to an actual value of the SUT, in order to refine the verdict. In this case, call to observations are automatically added at each step. The set of tests based on a given test selection criteria and a model is called a *test suite*.

## 3. The TOCL Language

We now present the TOCL property language and its different constructs. We then present the formal representation of the properties as automata that

| | **Predicate** | **REQ/AIM** |
|---|---|---|
| 1 | `self.current_user.isOclUndefined()` | `@AIM:BUY_Login_Mandatory` |
| 2 | `not(self.current_user.isOclUndefined()`<br>`and let tm... in tm.available_tickets = 0` | `@AIM:Buy_Sold_Out` |
| 3 | `not(self.current_user.isOclUndefined()`<br>`and let tm... in not(tm.available_tickets = 0)` | `@AIM:Buy_Success` |

Figure 4: Test targets computed from operation buyTicket

| Step | Operation | Behavior |
|---|---|---|
| 1 | `sut.login(REG_USER, REG_PWD)` | `@AIM:LOG_Success` |
| 2 | `sut.buyTicket(TITLE1)` | `@AIM:BUY_Success` |

Figure 5: Test case covering Target 3

can be exploited in a testing process.

### *3.1. Property Pattern Language*

The property description language is a temporal extension of OCL. This language is based on patterns which avoid the use of complex temporal formalisms, such as LTL or CTL. We ground our work on the initial proposal of Dwyer *et al.* [5]. In a study reported in the paper, the property patterns were able to cover 92% of existing properties, with a complete semantical equivalence w.r.t. LTL. Besides, the authors showed that these property patterns considerably simplified the expression of such properties w.r.t. to temporal logics, such as LTL.

We consider that 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, that describe the absence, occurrence, or succession of events:

(*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/at most/exactly $k$ times),

(*iv*) $E_1$ `precedes` $E_2$ means that event $E_1$ (directly) precedes event $E_2$,

(*v*) $E_1$ `follows` $E_2$ means that event $E_2$ is (directly) followed by event $E_1$.

A variant of the `precedes` (resp. `follows`) patterns exists to specify that event $E_2$ is `directly` preceded by (resp. `directly follows`) event $E_1$. Notice that the difference between these two patterns is subtle: $E_1$ `precedes` $E_2$ means that $E_2$ should not occur if $E_1$ has not occurred. However, it is possible that $E_2$ does not occur after having observed $E_1$. On the opposite, $E_2$ `follows` $E_1$ specifies that, once $E_1$ has occurred, $E_2$ has to occur. However, $E_2$ does not require $E_1$ to occur.

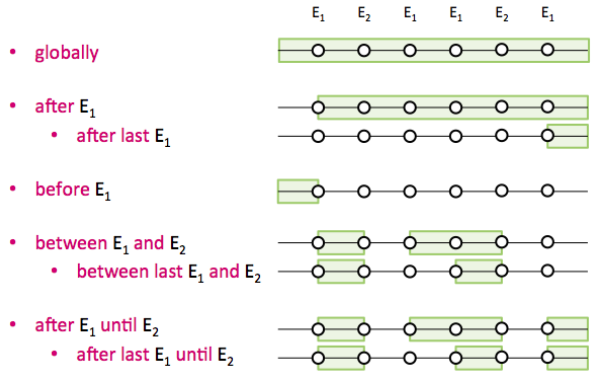



Figure 6: Graphical representation of the scopes

*Scopes.* Five scopes (and three variants), shown in Fig 6, can apply to a temporal pattern $TP$:

(*i*) `globally` means that $TP$ must be satisfied on any state of the whole execution,

(*ii*) `before` $E$ means that $TP$ must be satisfied before the first occurrence of $E$,

(*iii*) `after` $E$ means that $TP$ must be satisfied after the first (resp. last) occurrence of $E$,

(*iv*) `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*) `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 $E_2$.

Variants on `after` and `between` scopes have been introduced to consider the `last` occurrence of their respective opening events.

*Events.* In Dwyer's seminal paper [5], events are not precisely defined, and left to the user's discretion. Based on our context, we have proposed that 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 an 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 requirement can be expressed by the following three properties that 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})* | |
| ***and*** | *isCalled(logout, {@AIM:LOG_Logout})* | (*Property* 2) |
| ***never*** | *isCalled(buyTicket, {@AIM:BUY_Success})* | |
| ***after*** | *isCalled(logout, {@AIM:LOG_Logout})* | |
| ***until*** | *isCalled(login, {@AIM:LOG_Success})* | (*Property* 3) |

*Property* (1) *specifies that before a first successful login, it is not possible to sucessfully buy ba ticket. Property* (2) *specifies that when the user is logged in, he may buy a ticket. 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.*

These patterns simplify the specification of temporal properties which does not require the validation engineer to master temporal logics such as LTL or CTL. This property language makes it possible to express both safety ("something bad never happens") and liveness ("something good keeps happening") properties, depending of the combination of scope/pattern that is used. In this work, we do not consider the test properties that express a liveness property. For example, the combination of the `globally` or `after` scopes and the `existence` pattern, describing a liveness property ("after some event, there exists eventually another event") are not targeted by this work. Indeed, such combinations, that are not bounded by an event of the future can not be properly exercised by a testing approach.

### *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 $\omega$-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 [10]. 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 $\Sigma$ be a set of events. An automaton is a quintuplet $\mathcal{A} = \langle Q, q_0, F, R, T \rangle$ 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 ($T \subseteq Q \times \mathcal{P}(\Sigma) \times Q$) labelled by a set of events.*

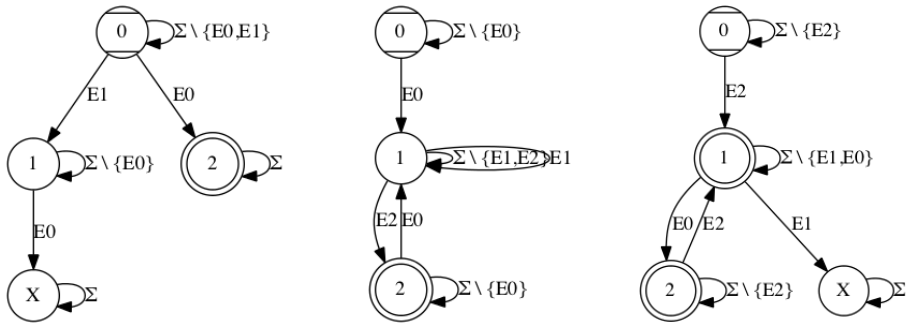
We call $\alpha-$transitions the transitions of $T$ that are labelled by the events expressed in the original property, and we call $\Sigma-$transitions the other transitions. $\Sigma-$transitions are named after their expression as they are labelled by a restriction on $\Sigma$ (the set of all possible 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. 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. Finally, it is possible that the automaton does not display any rejection state, as some constructs may design a property that can never be violated (e.g. Property 2 in Figure 1).

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 7 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 occur 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.*




Property 1 Property 2 Property 3

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

Figure 7: Automata representation for the properties given in Example 1

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 sections explains 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 illustrate or exercise the property.

## 4. Test selection criteria for TOCL properties

We present in this section the automata coverage criteria that we propose. These dedicated coverage criteria are motivated by the fact that classical coverage criteria on automata are not relevant for our property automata:

- transition or transition-pair coverage criteria make no distinction between the transitions of the automata. In the case of our property automata, all the transitions are not of equal importance. For example, consider the automata provided in Fig. 7. Reflexive $\Sigma-$transitions only exist to capture all possible executions of the model but their sole purpose is to put the model in a state from which the $\alpha-$transitions can be activated. Classical coverage criteria would target these $\Sigma$-transitions, resulting in additional tests that are not necessary w.r.t. the property. Besides, such transitions may not be all reachable, depending on the underlying model, resulting in unfruitful computations.

- classical automata coverage do not take into account the origin of the transition (from the scope or from the pattern) of the initial property.

Thus, these criteria do not provide a fine-grained coverage of specific parts of the property.

- classical automata coverage criteria are focused on covering the transitions of the automata and thus they can only be used to illustrate a nominal interpretation of the property. As a rejection state may exist, it could be relevant to provide a means to target these states, in an attempt to violate the property.

To address these issues, we propose new coverage criteria, focused on $\alpha-$transitions, aiming at activating them, but also focusing on different paths which iterate over specific paths in the automaton. This first set of dedicated coverage criteria address nominal cases described by the property. In addition, we also provide a robustness coverage criterion that targets corner cases of the property. In this section, 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*

An abstract test case is defined on the model as a finite sequence of steps, 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. Nominal Coverage Criteria for the Property Automata*

Nominal coverage criteria aim to illustrate the property that is described. In practice, these criteria expect the test cases to activate the sequences of events that are identified in the property. Notice that, since the model is expected to

satisfy the property, the paths inescapably leading to the error state are not supposed to be activable, 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 $\alpha$-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 [11]. 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. 7, 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 $1 \stackrel{E1}{\rightarrow} 1$. Also, another test case should illustrate the fact that two iterations of the scope are possible, by covering transition $2 \stackrel{E0}{\rightarrow} 1$.*

**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. 7, for Property 2, a test suite satisfying the $\alpha-$transition coverage criterion ensures the coverage of the following pairs: $(0 \stackrel{E0}{\rightarrow} 1, 1 \stackrel{E1}{\rightarrow} 1)$, $(0 \stackrel{E0}{\rightarrow} 1, 1 \stackrel{E2}{\rightarrow} 2)$, $(1 \stackrel{E1}{\rightarrow} 1, 1 \stackrel{E2}{\rightarrow} 2)$, $(1 \stackrel{E2}{\rightarrow} 2, 2 \stackrel{E0}{\rightarrow} 1)$, $(2 \stackrel{E0}{\rightarrow} 1, 1 \stackrel{E1}{\rightarrow} 1)$ and $(2 \stackrel{E0}{\rightarrow} 1, 1 \stackrel{E2}{\rightarrow} 2)$. A test suite satisfying this coverage criterion thus ensures 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 following two coverage criteria 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 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. 7, 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* $1 \stackrel{E1}{\rightarrow} 1$.

**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. The $k$-scope criterion is not applicable to all scopes, its application is restricted to repeatable ones, namely `between` and `after`.

**Example 6 ($k$-scope coverage).** *On the example shown in Fig. 7, 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* $1 \stackrel{E1}{\rightarrow} 2 \stackrel{E0}{\rightarrow} 1$.

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.

### *4.3. Robustness Coverage Criteria*

The nominal automata coverage criteria 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 also be interesting to produce test cases that attempt to violate the property. The property violation is 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.

To achieve that, we propose to weaken the events that label transitions leading to the error states, so as to potentially attempt to violate the property. Therefore, we propose mutation operators that apply to events so as to make them activable. As the model is expected to satisfy the property, we mutate this latter so as to characterize executions of the model (i.e., model-based tests) that lead to an error state of the automaton. Thus, we define a robustness coverage criterion that aims to simulate an erroneous implementation of the property that would allow the forbidden event to be activated.

*4.3.1. Event Mutation Operators*

The goal of this approach 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 [12], 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.

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

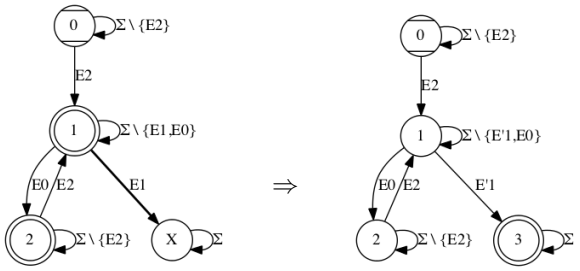
When applied, this mutation also removes the postcondition and tags, in order to weaken the event, and increase the chances that the mutation will produce an activable event.

*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 applied to the precondition, this rule also removes the postcondition from the event.

**Example 7 (Event mutation).** *Consider the examples provided on Fig. 7, 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.*




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

Figure 8: Mutation of the automaton for Property 3

*4.3.2. Automata Mutation and Robustness Coverage Criteria*

The mutation operators that we propose can be applied on a given property automaton $\mathcal{A}$. 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 $\mathcal{A}'$ the new automaton obtained after mutation.

**Example 8 (Automaton mutation).** *Figure 8 displays the application of a mutation on the automaton associated to Property* 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:*

| Step | Operation | Expected behavior |
|---|---|---|
| 1 | `sut.buyTicket(TITLE1)` | `@AIM:BUY_Error_Login_First` |
| 2 | `sut.login(REG_USER,REG_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 a success and 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(REG_USER, REG_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 to be performed successfully.*

## 5. Property-Based Testing with TOCL

We describe in this section the possible uses of the coverage criteria that were presented previously. This approach has been initially tool-supported and experimented in the context of the TASCCC project, during which a dedicated tool has been developed as an Eclipse plug-in[1]. Since the end of the project, this tool has been transferred to the Smartesting Solutions & Services company, and it is now integrated as a plug-in to the CertifyIt test generator (Figure 9).

### *5.1. Test Suite Evaluation*

The coverage of a property can be measured using its associated automaton. Each step of each test is matched (according to Definition 2) with a transition of the automaton. When replaying the abstract test cases on the model, it is possible to evaluate which transitions of the property automata have been covered,

[1] A demo video of this prototype is available at: http://vimeo.com/53210102

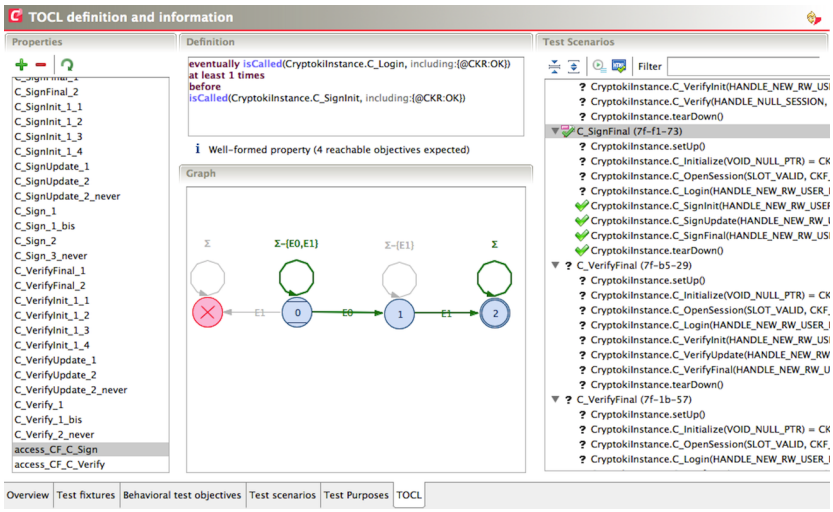



Figure 9: Screenshot of the TOCL plug-in in the CertifyIt test generator

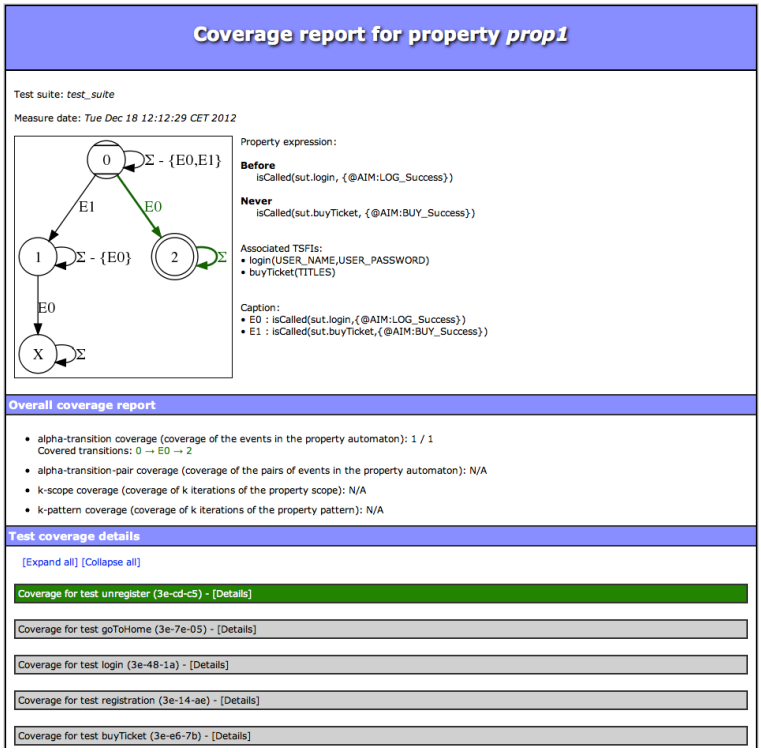



Figure 10: A test suite coverage report for Property 1

and, by that, which coverage criteria are satisfied. This coverage measure can be exported to be presented to the validation engineer as a web page indicating:

- the considered property,
- the automaton, on which covered transitions are distinguished,
- the summary on the satisfaction of the test selection criteria
- the detail, for each test, of which transition is covered at each step.

Figure 10 shows a report that is produced after the analyze of one of the properties of the example. In case of robustness test cases, the report shows which mutations are possible on the property automaton, and among them, which ones have been covered by the test suite, and the detail for each test.

For a given property, test cases can be classified into three categories.

($i$) The test may reach (one of) the final state(s) of the automaton, which makes it relevant, as it contains a sequence of events that is described in the property.

(*ii*) The test may never reach any final state, and thus, the test case is considered to be irrelevant w.r.t. the property.

(*iii*) The test may reach an error state. This case is not supposed to happen if the model respects the property. However, its existence reports a violation of the property, and the test provides a counter-example showing that there is a non-conformance between the model and the property. Two reasons may explain this issue. The error may come from the model, which is too permissive w.r.t. the property, or the property that is too restrictive w.r.t. the system. In both cases, this information helps the validation engineer either to correct the model (which is used to compute test cases) or the property (which is also used to produce complementary test cases).

Once the property coverage has been measured, it is possible to use the automaton to generate additional test cases that aim at improving the targeted coverage score.

### *5.2. Test scenarios generation*

The test scenario generation that is proposed relies on the scenario-based test generator provided by CertifyIt. This feature, described in [13] is named "Test Purposes" and makes it possible, for a validation engineer, to write test scenarios. These latter describe sequences of operations, with intermediate states (characterized by a predicate).

Figure 11 illustrates the syntax of the test scenario language. It is composed of structures that are voluntarily close to the natural language, so as to be easily adopted by the validation engineers. Notice that the Test Purpose language allows to write partial scenarios, in which all the operations do not have to be specified. Instead, the user can specify that, at some point, "any operation can be used", "any number of times", until reaching a given model state, or activating a given operation. The semantics of this language is close to regular expressions. It is expected that the tests resulting from this scenario will match this expression.

Such a scenario is unrolled and processed by the CertifyIt test generation engine, which computes automatically the unspecified parts (corresponding to the *any_operation any_number_of_times*) and replaces them with concrete operation calls that instantiate the test scenario.

For our approach, from the TOCL property automata, we have developed a scenario generator that identifies the transitions that have to be covered (de-

**use** *any_operation any_number_of_times* **then**
**use** sut.buyticket **to_reach** "self.currentUser.isOclUndefined()" **on_instance** "sut" **then**
**use** *any_operation any_number_of_times* **then**
**use** sut.login *to_activate behavior_with_tags* {AIM:LOG_Success}

Figure 11: Example of test scenario

pending on the selected coverage criterion), and creates one scenario per transition to be covered. This step relies on a modified version of the McNaughton-Yamada algorithm [14] to compute regular expressions from an automaton. Our version generates a regular expression on events that provides the path expressions for the considered automaton, ending by a transition that leads to one of the final states of the automaton. Once a regular expression has been computed, its translation into a "test purpose" is straightforward.

Figure 11 provides an example of the scenario that would be produced to test the robustness of Property (1), in which the test attempts to perform a ticket buying, before login in. An example of instantiation of this scenario is thus given in Example 8.

### *5.3. Model Validation*

By extension, the property coverage measure and test generation can be used as a means to partially validate the model, by testing it on a set of relevant executions.

As explained previously, the property automata may display error states, i.e. states in which the property is violated. On the example on Fig. 7, Property (1), state X represents a state that can be reached when the model authorizes a sequence in which a successful *buyTicket* is followed by a successful *login*. When replaying the tests computed from the model on the automaton, if this state is reached, a non-conformance between the model and the property is detected.

From there, the error can be at two levels, that need to be investigated by the validation engineer:

- First, the error can be in the model, meaning that this latter is too lax and does not respect the property that was written. In this case, the consequences can be severe, as the model may have already been used to produce test cases, that may already have been executed to validate an implementation.

- Second, the error can be in the property that is incorrectly written and too restrictive.

The other way to validate the model uses the test generation phase. This latter consists in finding automatically a sequence of operation calls that, from the initial state, reaches the considered test targets. The test generator can thus be used with two different objectives.

- First, it is possible that the test generator fails to compute a test case, because this latter represents a transition of the automaton (i.e. a test target) that can not be activated on the model. This indicates that either the model is too restrictive, as it does not allow to perform certain sequences of operations, or the property is incorrectly written, as it expects a behaviour that is not allowed by the model.

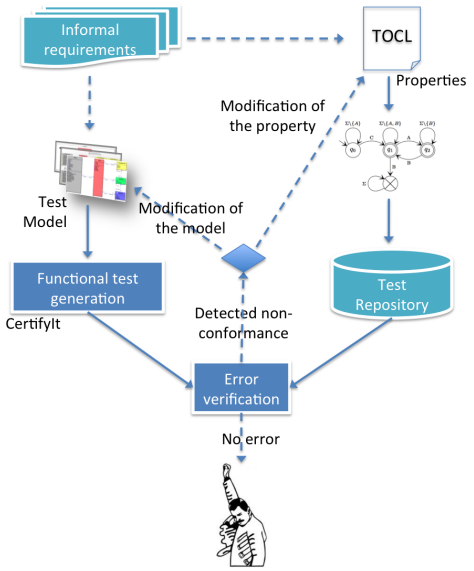



Figure 12: Model validation process using TOCL properties

- Second, the CertifyIt TOCL plug-in integrates a special mechanism that aims at targeting the error states of the automata to look for a counter-example to the property. If the model is correct, these targets should not result in a test reaching them. Otherwise, the test case represents a counter-example to the validity of the property on the model.

Notice that, for both cases, as the reachability problem is undecidable in the general case, the test generator uses a bounded exploration of the state space. It is thus possible that the test target is reachable within a deeper depth. However, the results that are produced can give a preliminary feedback to the validation engineer.

Figure 12 summarizes the process of model validation using the TOCL properties. TOCL properties are designed based on the informal requirements of the considered system. These properties are evaluated on the tests of the test repository. This latter is composed of tests generated by the usual functional approach as described in Sect. 2.3. However it can also be composed by tests that are produced by considering TOCL properties (to avoid overloading the figure, only the first case is depicted). If an error state is reached, the feedback that is given by the analysis of the test case, the model and the property, leads to the correction of the model or the property. The process works iteratively and incrementally.

During the application of this approach in research projects, we frequently faced a misuse of the property. The validation engineers who worked with TOCL often used this language as a means to express a test scenario (e.g. *"I*

*want to see a test in which operation A precedes operation B"*). However, the sequencing they described were rarely applicable for all the possible executions of the model, leading to the design of incorrect properties that, when confronted with the tests, raised errors (by reaching error states in the automaton). To overcome this kind of issues, we have proposed an assistant for the design of TOCL properties, that is now described.

## 6. Assistance to Property Design

In our experiments, we have noticed the external users of the TOCL language had some difficulties in understanding the correct semantics of each constructs. Thus, we noticed several errors that were frequently encountered when TOCL properties were written. Such errors may lead to severe consequences in the testing process.

To help the validation engineers who want to use TOCL, we have proposed a tool-supported assistance that helps writing the appropriate TOCL properties w.r.t. the test intention they had in mind. Before presenting the assistant and its underling principles, we first illustrate how misleading the semantics of TOCL can be.

### *6.1. Issues with the TOCL property patterns*

For example, consider the following informal requirement:

*"before buying a ticket, the user has to successfully log in"*

Intuitively, one may use the following construct:

**before** isCalled(sut.buyTicket, @AIM:Success)
**eventually** isCalled(sut.login, @AIM:Success)

Even though this construct seems to be straightforward from the informal requirements, it is not as correct as it seems. Firstly, this property considers as correct a sequence in which the user logs in, then logs out and finally buys a ticket. As a consequence, the informal requirement itself is not properly checked. Secondly, this property only describes the first occurrence of the ticket buying. Is it possible to buy a second ticket in a row or should the user re-authenticate each time? Thus, the informal requirement is not completely described, and it would require additional properties to be used in order to accurately formalize it.

To overcome this issue, we have proposed to assist the user in writing temporal properties using the appropriate patterns. To this end, we provide a decision tree, based on frequent requirement wording, that helps the user selecting the appropriate (set of) temporal properties that captures at best what (s)he intended.

*6.2. Assistance to the Validation Engineer*

We propose to assist the validation engineer in designing TOCL properties. To achieve that, we have started by 4 sentences that express a frequent informal requirement, based on symbolic events X, Y Z:

- to execute Y, X must be executed first
- to execute Y, X should not be executed
- between X and Z, Y must be executed
- between X and Y, Y should not be executed

These abstract sentences represent the starting point, from which additional questions will be asked to the user to refine its mind. In the end, depending on the different answers given at the different stages, some property patterns are proposed, based on X, Y, and Z, that the user only has to replace with actual TOCL event (e.g. *isCalled* events).

To save space, we only give here an instance of these questions, by unrolling the example: "to buy a ticket [Y], the user has to successfully log in [X]", which corresponds to the first formulation. Notice that some of these questions are also used for the other constructs. The questions that are considered are the following.

Q1. *Does X [login] directly precede Y [buyTicket] ?*
This question checks if there is a direct succession of event X followed by Y.

Q2. *Can several executions of Y [buyTicket] occur after X [login]?*
This question checks if X provides the privileges to execute Y only once (and thus it has to be called each time one wants to execute Y), or if the privileges given by X are not limited (except by another operation that will be identified in Q5). In practice, this question determines the number of executions of Y that can be done after X.

Q3. *Can several executions of X [login] occur before Y [buyTicket]?*
Similarly to Q2, this question checks if X can be repeated several times, or if only one execution of X is enough to execute Y.

Q4. *Is it mandatory to execute Y [buyTicket] after X [login]?*
This question will be used to determine the minimal number of execution of X in the considered scope.

Q5. *Does it exist an event Z that cancels the privileges given by X [login] w.r.t. Y [buyTicket]?* This question is used to determine the scope of the properties. If Z does not exist, the scopes only consider X. If Z exists, the intervals X – Z or Z – X are considered.

06. *Is it possible to iterate the whole sequence?* This question is used to determine if the generated scopes have to consider that several intervals can be observed, or if the satisfaction of the pattern is always unique in a given sequence.

On our example, the answer to the different questions produce the following set of TOCL properties:

- never *buyticket* before *login*
- *login* precedes *buyticket* before *logout* (Q1-No, Q5-Yes)
- eventually login at least 1 times between logout and buyticket (Q1-No, Q3-Yes, Q5-Yes)
- eventually *buyticket* at least 0 times between *login* and *logout* (Q2-Yes, Q4-No, Q5-Yes)
- never *buyticket* after *logout* until *login* (Q5-Yes, Q6-Yes)
- never *logout* between *login* and *buyticket* (Q5-Yes)

### *6.3. TOC-heLp - an Assistant for designing TOCL Properties*

We have implemented this assistant in a web application. This tool lets the user choose between a set of informal requirements that he wants to test. Subsequently, several questions will be asked to the user, in order to guide the decision on the combinations of scope/pattern that have to be used.

Figure 13 presents an overview of the tool[2]. The user can choose the informal pattern in the top part. Then the associated questions appear on the left-hand side of the page. Each question can be answered by "Yes" (it then becomes green) or "No" (it then becomes red). Simultaneously, the patterns correponding to the current set of answers are shown on the right-hand side of the page. In addition, it is possible to caption the X, Y, and Z symbolic events, to replace them with keywords (e.g. "buyticket") or TOCL events (e.g. "isCalled(sut.buyTicket,@AIM:BUY_Success)") so that the properties are directly expressed in the correct syntax.

We now describe the experiments that we performed to valide the approach on a realistic case study.

## 7. Experimental assessment with PKCS#11

We present in this section the experimental assessment of the proposed approach. First, we describe the PKCS#11 case study, for which we present the research questions and we describe the experimental settings and results we obtained. Second we provide additional reports on two research projects in which TOCL properties were used.

[2]available at: `http://projects.femto-st.fr/sites/femto-st.fr.mbt_sec/files/content/TOC-heLp/tochelp.html`

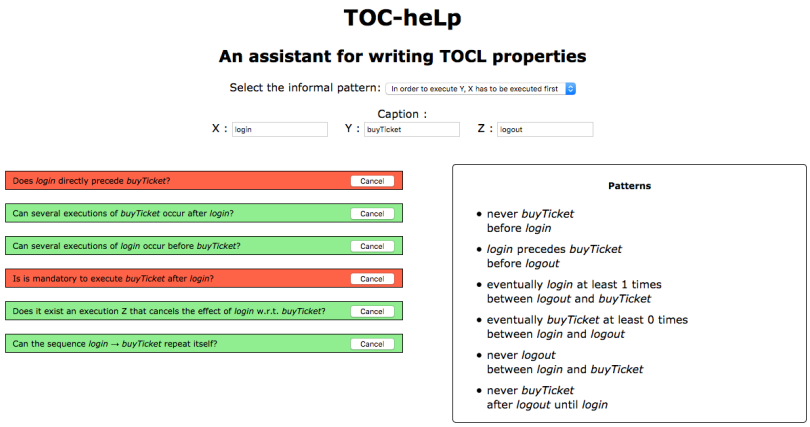



Figure 13: The TOC-heLP web application

### *7.1. The PKCS#11 case study*

RSA Public Key Cryptography Standards (PKCS) propose various standards to promote interoperability and security. Our study focuses on the PKCS#11 V2.20 specification (the official version published in 2004), which defines the *Cryptoki* interface, an API for cryptographic hardware, such as HSM or smartcards. The adoption of this standard for communicating with cryptographic tokens in the industry is nearly omnipresent, even though other complementary interfaces are offered by the security tokens.

Shortly, an API based on the PKCS#11 specification *initiates the communication* with the token before any other function call. Then, in order to perform cryptographic functions, such as signing a message, it *opens a session* and *logs* the user. When a function is called in the token's API with a reference to a specific *object* (for instance a key used for signing a message), the token first checks the permissions of the object in order to allow the usage of the function. Permissions are *attributes* that might be represented as boolean flags representing the properties of an object (for example CKA_SIGN flag of a cryptographic key indicates whether a key can be used for signing a message). Further, accesses to operations and objects are controlled through the interface. In general, to perform cryptographic operations, the user must *log in* to the application. To guarantee security, *Cryptoki* implicitly or explicitly defines security requirements that must hold. Most of these requirements can be assimilated to sequencing properties (e.g. *"a signature verification operation must have been initialized"*).

Thus, this case study is relevant to our approach and representative of the kind of application it addresses.

#### *7.1.1. Research Questions and Experimental Procedure*

PKCS#11 is a standard for encryption that is representative of the systems that our approach targets. Indeed, it requires commands to be performed in a

row, and thus, it is relevant to describe operation sequences using TOCL.

The research questions we would like to address with this case study are the following:

- To what extent is TOCL relevant for expressing sequencing properties?
- To what extent is the TOC-heLp assistant useful for designing relevant properties?
- To what extent do TOCL properties help validating the model?
- To what extent does this property-based testing approach improve the quality of a test suite?

To address these questions, we have designed the following experiment.

- First, a trained validation engineer is asked to write some TOCL properties from the informal security requirements of the standard. We thus evaluate: the ability to use the language, how many requirements have been formalized using TOCL properties, and the impact on the initial model, w.r.t. the usual approach that consists in instrumenting the model by introducing artificial test targets that pollute the model code.
- Second, we ask the same user to use the TOCL-helper to consider the same security requirements and derive TOCL properties. We compare the produced TOCL properties with the manually designed ones.
- Third, we perform an evaluation of an existing functional test suite, computed using the Smartesting CertifyIt tool, completed by complementary tests that can be either automatically computed or semi-automatically designed (by means of test purposes) by the validation engineer.
- Finally, we evaluate the error detection capabilities of the test cases that were produced.

*7.1.2. PKCS#11 Model*

In an MBT approach, test requirements (functional and security functional ones) are commonly identified from a defined testing perimeter, on the basis of the specifications and available documents. Thus, our case study relies on a subset of the PKCS#11 specification, which, based on industry experts opinion, was qualified as self-contained, realistic and sufficient to illustrate the main aspects of the specification and as well to illustrate the use of model-based testing for such security components. Classically, in the industry, security tokens support only sub-parts of PKCS#11. Thus, the considered perimeter of the study are 24 functions most commonly present in the tokens: general purpose, session, token, key and user management functions, as well as cryptographic functions for digesting, signing messages and verifying signatures.

We designed a UML/OCL model for PKCS#11, covering a set of 24 functions. Figure 14 depicts a simplified class diagram of the PKCS#11 model,

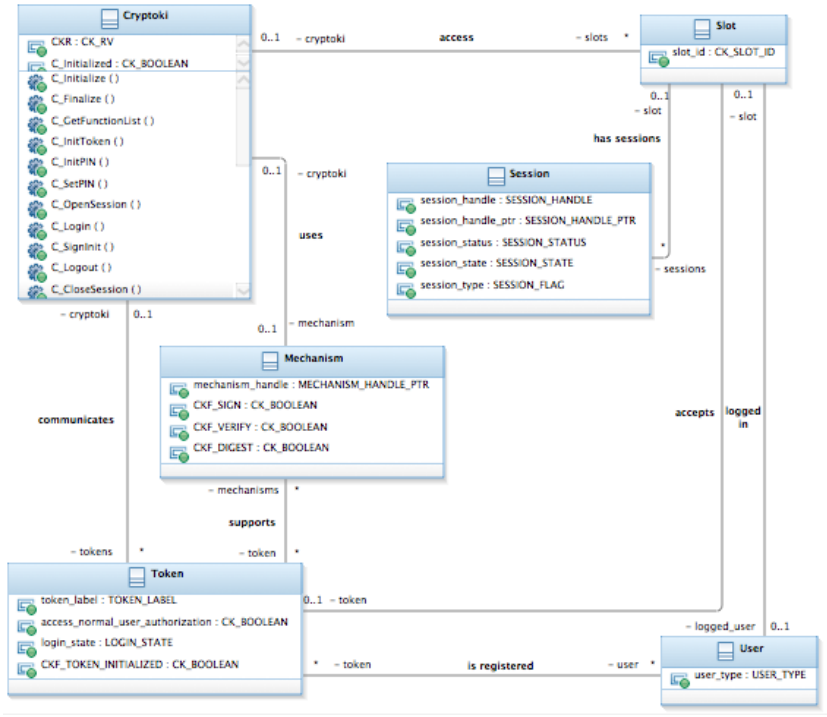



Figure 14: PKCS#11 test model

which contains six classes: *Cryptoki, User, Token, Slot, Session, Mechanism.* We represent the API *Cryptoki* that offers to a *User* an interface for communicating with cryptographic tokens, modeled by the class *Token.* Each token is connected to the system through a *Slot.* Finally, once the user has been connected to a *Session*, *Cryptoki* offers cryptographic operations, such as signing a message (e.g. function "C_Sign") or verifying a message signature (e.g. function "C_Verify"), with different cryptographic algorithms, represented by the class *Mechanism.* Figure 15 shows the OCL postcondition of the C_SignInit command.

Table 1 summarizes the functional and security functional requirements of the PKCS#11 case study, according to the groups of functions defined by the PKCS#11 specification. The total number of functional requirements (FR) for the considered subset of PKCS#11 is 158. In addition to these requirements we have identified 49 security functional requirements (SFR).

### *7.1.3. TOCL Properties Design*

50 TOCL properties were designed to address the Security Functional Requirements that could be handled by temporal properties. To reach this number, we relied on the expertise of the validation engineer, who wrote nearly 40% of these properties. The remaining 60% were produced using the TOC-heLp assis-

```
---@REQ:C_SignInit
if self.initialized = false then
   ---@AIM:CRYPTOKI_NOT_INITIALIZED
   result = CKR_CRYPTOKI_NOT_INITIALIZED
else
   if session.loggedUser.oclIsUndefined() then
      ---@AIM:USER_NOT_LOGGED_IN
      result = CKR_USER_NOT_LOGGED_IN
   else
      ---@AIM:OK
      result =  CKR_OK
   end if
end if
return result
```

Figure 15: OCL postcondition sample of C_SignInit

tant. Notice that this latter produced 12 properties that were already designed manually by the validation engineer. Also, in a few cases (3 properties), the assistant helped correcting an inappropriate TOCL construct that was proposed by the test engineer.

Especially, the assistant was used to systematize the design of sets of sequencing properties for security functions (signing, hashing, etc.) that all worked on the same principle. First, they have to be initiated (e.g. SignInit), then an update has to be performed (e.g. SignUpdate). In the end, it is mandatory to terminate (e.g. SignFinalize) the command. Another possibility is, after the initiation of the security function, to call the function itself, once (e.g. Sign), which applies the crypto function in an atomic way. The TOC-heLp tool was thus very convenient to design properties expressing this chain of events, based on the informal pattern: *between X and Z, Y must be executed*. This latter was declined with triplets *Op*Init, *Op*Update, *Op*Finalize for each security operation *Op* among { Sign, Hash, Digest, Verify }.

We provide hereafter a sample of the properties that were designed for the

Table 1: PKCS#11 case study perimeter

| **Test Requirement category** | **#FR** | **#SFR** |
|---|---|---|
| general purpose | 7 | 4 |
| slot and token management | 22 | 5 |
| session management | 32 | 9 |
| object management | 6 | 2 |
| digesting | 28 | 9 |
| signing | 32 | 10 |
| verifying signatures | 31 | 10 |
| **total** | 158 | 49 |

PKCS#11 case study. These properties can be classified into 4 categories, for each we provide an example using a given cryptographic function (usually C_Sign). Notice that, in the experiment, similar properties have also been designed for the other cryptographic functions (C_Hash, C_Digest, C_Verify).

*Access rights.* The access to cryptographic functions of the component is restricted to identified users. Therefore, it is mandatory to login before being able to successfully execute such a function:

- **eventually** isCalled(C_Login, @AIM:OK) **before** isCalled(C_SignInit, @AIM:OK)
- **eventually** isCalled(C_Login,@AIM:OK) **between** isCalled(C_Logout, @AIM:OK) **and** isCalled(C_SignInit, @AIM:OK)

*Initialisation of cryptographic operations.* These functions necessarily have to be initialized to be successfully invoked.

- isCalled(C_SignInit, @AIM:OK) **precedes** isCalled(C_Sign, @AIM:OK) **globally**
- isCalled(C_SignInit, @AIM:OK) **precedes** isCalled(C_SignUpdate, @AIM:OK) **globally**

*Context-dependant invocation restriction.* In the PKCS#11 standard, it is not possible to invoke cryptographic functions in parallel. Thus, we have to make sure that no function can be re-initiated before its current execution is completed. Regarding cryptographic functions, it is possible either to invoke *CryptoUpdate* functions several times before calling *CryptoFinal*, or to invoke a single time the *Crypto* function itself (where *Crypto* is a cryptographic function, such as Sign, Hash, etc.)

- **never** isCalled(C_SignUpdate, @AIM:OK) **after** isCalled(C_Sign, @AIM:OK) **until** isCalled(C_SignInit, @AIM:OK)
- **never** isCalled(C_Sign, @AIM:OK) **after** isCalled(C_SignUpdate, @AIM:OK) **until** isCalled(C_SignInit, @AIM:OK)

*Termination of cryptographic operations.* Finally, it is mandatory to ensure that all initiated cryptographic operations are finalized.

- isCalled(C_SignInit, @AIM:OK) **precedes** (isCalled(C_SignFinal, @AIM:OK) **or** isCalled(_Sign, @AIM:OK)) **globally**

Table 2: PKCS#11 test suites metrics

| **Test Selection Criterion** | **#Test targets** | **#Test cases** | **Cov. in %** | |
|---|---|---|---|---|
| | | | **FR** | **SFR** |
| Structural | 206 | 184 | 100 | 40 |
| TOCL | 311 | 90 | 31 | 58 |
| Manual | 24 | 24 | 45 | 16 |

Table 3: PKCS#11 test suites execution

| **Test Selection Criterion** | **#Test cases** | **Test execution** | |
|---|---|---|---|
| | | **#Failed Tests** | **#Distinct Faults** |
| Structural | 184 | 6 | 5 |
| TOCL | 90 | 12 | 3 |
| Manual | 24 | 0 | 0 |

*7.1.4. Test Execution and Results*

In order to distinguish the test execution results, we designed a test suite corresponding to the functional test suite generated using CertifyIt, and a test suite corresponding to the tests generated from the TOCL properties. We also considered a manual test suite of PKCS#11 that was provided with the implementation that we considered.

Table 2 summarizes the results of this evaluation. As given in the table, the manual tests cover barely 45% of the functional requirements, showing thus the incompleteness of the manual test suite, compared to the automatically-generated tests, with respect to the specification.

In addition, the TOCL tests on their own are not sufficient to cover the FR, as the structural criteria are not sufficient to cover the security functional requirements (SFR). The coverage measure notified that the manual test suite covers 14 TOCL properties, which represent about 16% of the SFR. We evaluated the number of *distinct faults* revealed by the failed tests. As several tests can reveal the same fault, we were specifically interested in the diversity of the detected faults by each test selection criterion, that we refer to as *distinct faults*. We report these results in Table 3, showing that both test selection criteria detected a panel of faults, while the manual test suite did not reveal any fault.

In addition, Table 4 details the implementation (SoftHSM) discrepancies with respect to the specification detected by the failed tests. For each one, we show the function, the expected return code and the actual code returned by the function during the execution of the test. Results from Tables 3 and 4 show that each test suite reveals different discrepancies complementary to the other test suites, thus increasing the detection of distinct faults. We see that there is no intersection between the different faults detected by each test suite, illustrating a real complementarity between the two approaches.

Table 4: SoftHSM discrepancies with PKCS#11 specification

| Function | Expected output | Actual output | Test suite | |
|---|---|---|---|---|
| | | | Structural | TOCL |
| C_Logout | CKR_USER_NOT_LOGGED_IN | CKR_OK | X | |
| C_DigestInit | CKR_USER_NOT_LOGGED_IN | CKR_OK | X | |
| C_DigestInit | CKR_OK | CKR_OPERATION_ACTIVE | | X |
| C_SignInit | CKR_OK | CKR_OPERATION_ACTIVE | | X |
| C_SignUpdate | CKR_USER_NOT_LOGGED_IN | CKR_OK | X | |
| C_Sign | CKR_USER_NOT_LOGGED_IN | CKR_OK | X | |
| C_SignFinal | CKR_USER_NOT_LOGGED_IN | CKR_OK | X | |
| C_VerifyInit | CKR_OK | CKR_OPERATION_ACTIVE | | X |

*7.1.5. Conclusions of the Experiment*

We answer here the research questions that were considered.

- *To what extent is TOCL relevant for expressing sequencing properties?*
  Before introducing TOCL, the validation engineers had to manually encode the sequencing properties in the model, using ghost variables to encode the underlying automaton. To test these situations, the model was originally polluted with additional branches in the code, so as to create "artificial" test targets that can be considered by the functional test generator. TOCL thus considerably simplifies the model, as it removes pieces of code that were, in practice, only used for driving the test generation phase. Besides, the test generation process is externalized and thus, several coverage criteria can be proposed to test more or less extensively the considered property.

- *To what extent is TOC-heLp assistant useful for designing relevant properties?*
  As reported in the experiment, the use of the assistant helped producing a significant set of test properties that were not considered by the test engineer. Besides, these properties helped detecting incorrect constructs used by the validation engineer. The experiment has shown that this tool makes it possible to systematize the generation of sequencing properties. Thus it is helpful to the validation engineer as it unburdens him/her from having to consider all the different situations that may occur for a given informal properties that (s)he wants to test.

- *To what extent do TOCL properties help validating the model?*
  During the design phase of the model, the TOCL properties help detecting inconsistencies between the model and some of the requirements, especially in terms of model variables describing the sequencing of cryptographic operations (digest, sign and verify), that were incorrectly updated.

- *To what extent do the proposed TOCL coverage criteria improve the quality of a test suite?*
  The experiments have shown that the initial functional test suite did not achieve an acceptable coverage of the properties. As a consequence, the

nominal situations described in the properties were not tested, and potential fault in them would be missed. We have seen that the TOCL tests complement the functional tests as they were able to detect faults that were present in the considered implementation of PKCS#11, while being undetected by the functional test suite, and vice-versa.

### *7.2. Other Experiments with TOCL*

TOCL has been used in other projects that revealed its usefulness. We summarize here the context of these experiments and the conclusions that can be drawn from them.

#### *7.2.1. Experiments on GlobalPlatform during the TASCCC project.*

GlobalPlatform is an industrial standard for managing resources for multi-application smartcards. It describes all the functionalities and interfaces for managing the administrative aspects of a card all along its life cycle. An important fact related to the GlobalPlatform standard is that it is designed to allow different actors (phone companies, banks, transportation operators, etc.) to coexist on the same card. Such a possibility is offered by the notion of a Security Domain (SD) that represents an application through which all interactions with the operating system are performed.

During the TASCCC project, we focused on GP UICC profile, and specifically on the life cycle of the card, which is expected to comply with a simple state machine displaying 5 states. The OP_READY and INITIALIZED states both indicate that the card is ready to receive commands from the issuer, but not from the card holder. State SECURED means that the card is ready to receive commands from the card holder. If a security violation happens, the card goes to the CARD_LOCKED state. Finally, when the card is TERMINATED, no command can be successfully invoked. The life cycle of the card is controlled by the applications, which use the *"setStatus"* operation to set the life cycle state, accordingly to the state machine.

This project was aimed to automate coverage reports in the context of Common Criteria (CC for short) [15] certifications. This certification process consists, for a manufacturer, to provide evidences that the development of the security product, here the smart cards, has been done following different guidelines. Depending on the Evaluation Assurance Level (EAL) that is targeted, more or less detailed information has to be provided to the evaluator. TOCL was used to evaluate the test cases produced by validation engineers at Gemalto (the smartcard manufacturer involved in the project) and to automatically produce coverage reports of the property. As a proof of concept, we thus designed 4 properties that were intended to cover this life cycle, and we generated dedicated reports for the CC evaluation, saving time for the industrials and efforts for the evaluator who did not have to relate each test to the functional security requirement that it was intended to cover.

The Common Criteria evaluator of the TASCCC project reported that (extract from evaluation report [16]):

*It has been validated that the produced tests fully satisfy the usual evaluation criteria applied for this kind of product [i.e. smart cards]. One of the most important criterion is the relevance of the test cases, especially when automatic tools are used. The study shows that the test cases of the TASCCC campaign have the same level of relevance as test cases that would have been manually produced by a validation engineer. The advantage of this approach is to produce more tests and thus exercise the product in additional various contexts.*

### *7.2.2. Use TOCL with a SCM Case Study*

The second usage of TOCL we report here was realized during a national research project named MBT_Sec (*MBT for security components*) that involved the French DGA[3]. The experimentation was performed in the context of an evaluation process of cryptographic products. The agency requires that these products have a qualification issued by a national authority, the French Network and Information Security Agency (*ANSSI*). This qualification ensures the robustness of the security product against attackers of a defined skill: it indicates that the product can protect information of a given sensibility level (potentially classified information), under specified conditions of use. In this context, the evaluation of cryptographic software supplies to the authority in charge of the qualification all the technical elements needed for this assurance. This evaluation focuses in particular on the ability of the product to ensure information availability, confidentiality and integrity.

Our experimentation focused on a cryptographic library we call SCM for *Software Cryptographic Module*. This library offers classical cryptographic services like symmetrical and asymmetrical encryption, digital signature, hash computing and random generation. It embeds an internal sequencing controller which maintains a coherent state of the module in any state of the system. An objective of our work was to address the underlying state-machine which can not be manually validated due to its complexity (more than a thousand states and sixteen thousand transitions). Initially, this automaton was modelled using 11 additional classes and 38 operations per class. This represented 418 operations and additional 2269 lines of OCL to guide the test generation engine, which resulted in a very time consuming maintenance of the model.

Rather than exploiting this hardly maintainable model structure, the TOCL mechanism allowed to systematize the testing phase. Indeed, the states of this automaton are determined by the values of a given set of flags. For each command, a specific flag can be requested or forbidden, in order to execute the command. Consequently, the command may set a flag, reset it, or invert it. Thus, we defined, using the TOC-heLp assistant, 7 templates of TOCL properties that check the implementation of the commands sequencing, regarding the values of the considered flags. These templates are the following:

[3] Armament Procurement Agency

- there is no erasure of a flag between its last setting and its subsequent usage (as required).
- once a flag is set, and until it is erased, a command that requires this flag can be invoked.
- in order to execute a command that requires the flag to be set, it has to be set first.
- once a flag has been erased, it has to be set in order to execute a command that requires it.
- there is no setting of a flag between the last erasure of the flag and its use (as forbidden) to execute a command.
- once a flag is erased, and until it is re-set, a command that requests the absence of this flag can be invoked.
- once a flag has been set, it has to be erased in order to execute a command that forbids it.

Each template is then instantiated with a particular flag in order to provide a TOCL property. Notice that it is possible to ensure the traceability of the properties w.r.t. the considered flag and the test intention that is expressed by the considered template.

By considering the SCM table of 18 flags and 37 commands (describing 44 combinations of before/after flags), we were able to easily generate an exhaustive set of 935 TOCL properties. We evaluated the existing test suite on these properties to check if they were covered by the tests. Notice that each property could be documented by its informal expression, instantiated for the considered flag and appropriate commands, providing an interesting and useful feedback for the analysis of the coverage measure.

Due to confidentiality issues, it is not possible to provide further metrics for this experiment. However, we can draw the following conclusions:

- TOCL properties made it possible to simplify the initial UML/OCL model by externalizing a huge part of OCL (40%) that was initially only used to drive the test generation ;
- the templates provided by the TOC-heLp assistant were evaluated in collaboration with DGA experts who validated the templates that were produced by the tool.

These two experiments, and the feedback from our industrial partners on our approach, corroborate the conclusions of our case study on PKCS#11.

## 8. Related Work

In this section, we compare our model-based testing approach with the related works. First, we compare to other UML/OCL based test generation approaches. Second, we consider works on the use of properties for testing.

### *8.1. Model-Based Testing from UML/OCL*

Many approaches have considered the use of UML, coupled with OCL, to automatically generate test cases.

In [17] the authors propose the use of Higher-Order Logics to translate UML/OCL models and automatically generate test cases. This approach is similar to the function test generation described in Sect. 2.3. Similarly, a structural coverage is also considered in [18, 19] in which the authors decompose the OCL constraints into Disjunctive Normal Form. The resulting formulae are then solved by a constraint solver to generate test data activating the different behaviors described in the specification. Also, in [20], the authors consider search techniques to generate test cases. These works relate to the functional test generation approach proposed by the CertifyIt tool. However, the notion of properties is not considered and the coverage criteria that are considered focus on the structure of the OCL constraints. Our approach aims to complement such approaches by considering additional external entities

A lot of scenario-based testing works focus on extracting scenarios from UML diagrams, such as the SCENTOR approach [21] or SCENT [22] using statecharts. The SOOFT approach [23] proposes an object oriented framework for performing scenario-based testing. In [24], Binder proposes the notion of round-trip scenario test that cover all event-response path of a 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.

Close to our approach using mutations, the MoMut::UML tool [25] uses a model mutation approach to generate test cases that reveal the mutants produced by a set of mutation operators. Our approach differs in the sense that the mutations are performed on the property automata and not on the UML model, reducing the number of mutants that can be produced, and focusing the test cases on specific cases directly related to the property.

The RT-Tester tool [26] proposes various approaches to test from UML/SysML models. In particular, it is both able to provide a functional test generation strategy, and a requirement-based strategy based on LTL formulae that witness a given property. Our approach goes one step further as we also consider corner cases of the considered property. Besides, we do not rely on LTL but rather on a simpler temporal language that is easier to handle by the test engineers.

### *8.2. Property-Based Testing*

The notion of property-based testing is often employed in the test generation context. Several approaches [27, 28, 29] 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 [30]. 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. In [31], the authors define 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.

In [32], 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. Carvalho et al [33] propose to use controlled natural langauge to generate model-based tests. In this work, the requirements are expressed using a case grammar and test cases are derived from them. Our approach differs as the properties are in a semi-formal language, and they do aim to replace the model.

Based on Dwyer's work, jPost [6] uses a property expressed in a trace logic for monitoring an implementation. Similarly, in [34] 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. Also using a passive testing approach, [35] proposes to analyse event traces, coupled with test properties to validate distributed systems. Closed to our approach, [36] proposes a set of verification patterns, similar to Dwyer's patterns, that are used to generate test script templates, that the user has to complete. Contrary to our approach, this work does not consider different coverage criteria for each pattern, and the test generation phase is not automated.

In [37], the authors propose a property-based testing approach that relies on FsCheck, and a set of business rules models as input. We share a common vision of using properties for generting test cases and using them as a test oracle. However, the properties considered in this work do not take into account the dynamics of the system.

Recently, inspired from our work, an extension to OCL called OCLR [38] for "OCL for Run-time verification" was proposed by Wei et al. This work adds timing constraints to the same constructs that we initially published. However, the authors did not experience active testing using these properties, limiting their work to passive testing.

## 9. Conclusion and Future Work

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. This approach has been transferred into an industry-strength tool, and is now proposed as a plug-in to the CertifyIt test generator. 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. In addition, we have provided an assistant that can help the user to choose the appropriate TOCL construct, and makes it possible to systematize the design of test properties.

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 plan to improve the test generation engine, so as to be able to deal with multiple successive test targets, as this approach may produce. During the experiments we sometimes failed to generate automatically test cases that we were able to produce manually. One solution to do that, is to couple the symbolic test generation engine with search-based testing algorithms. Further, we plan to investigate the fault localization techniques, to help the validation engineer in determining the origin of the discrepancies than can be found when running the different test cases.

## References


[1] B. Beizer, Black-box Testing: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, Inc., New York, NY, USA, 1995.

[2] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Softw. Test. Verif. Reliab. 22 (5) (2012) 297–312.

[3] J. Warmer, A. Kleppe, The Object Constraint Language Second Edition: Getting Your Models Ready for MDA, Addison-Wesley, 2003.

[4] P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, A. Haddad, An access control model based testing approach for smart card applications: Results of the POSÉ project, JIAS, Journal of Information Assurance and Security 5 (1) (2010) 335–351.

[5] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, 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, 1999, pp. 411–420.

[6] Y. Falcone, L. Mounier, J.-C. Fernandez, J.-L. Richier, j-POST: a Java Toolchain for Property-Oriented Software Testing, Electr. Notes Theor. Comput. Sci. 220 (1) (2008) 29–41.

[7] B. Kanso, S. Taha, Specification of temporal properties with ocl, Sci. Comput. Program. 96 (P4) (2014) 527–551.

[8] S. Taha, J. Julliand, F. Dadeau, K. C. Castillos, B. Kanso, A compositional automata-based semantics and preserving transformation rules for testing property patterns, Form. Asp. Comput. 27 (4) (2015) 641–664.

[9] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, M. Utting, 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, 2007, pp. 95–104.

[10] K. Cabrera Castillos, F. Dadeau, J. Julliand, B. Kanso, S. Taha, A compositional automata-based semantics for property patterns, in: E. Johnsen, L. Petre (Eds.), iFM'2013, 10th Int. Conf. on integrated Formal Methods, Vol. 7940 of LNCS, Springer, Turku, Finland, 2013, pp. 316–330.

[11] J. C. Huang, An approach to program testing, ACM Comput. Surv. 7 (3) (1975) 113–128.

[12] R. A. DeMillo, Test adequacy and program mutation, in: ICSE, 1989, pp. 355–356.

[13] J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, F. Schadle, Model-based testing of cryptographic components – lessons learned from experience, in: ICST'13, 6th IEEE Int. Conf. on Software Testing, Verification and Validation, 2013, pp. 192–201.

[14] R. McNaughton, H. Yamada, Regular expressions and state graphs for automata, IEEE Transactions on Electronic Computers 9 (1960) 39–47.

[15] Common criteria for information technology security evaluation, version 3.1 (July 2009).

[16] D. Rouillard, Tasccc project - deliverable 5.4 - report on the integration of the ate requirements, Tech. rep., Serma Technologies (2012).

[17] A. D. Brucker, M. P. Krieger, D. Longuet, B. Wolff, A specification-based test case generation method for uml/ocl, in: Proceedings of the 2010 International Conference on Models in Software Engineering, MODELS'10, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 334–348.

[18] M. Benattou, J.-M. Bruel, N. Hameurlain, Generating test data from ocl specification, in: Proc. ECOOP Workshop Integration and Transformation of UML Models, 2002.

[19] L. V. Aertryck, T. Jensen, Uml-casting: Test synthesis from uml models using constraint resolution, in: AFADL'2003, 2003.

[20] S. Ali, M. Z. Iqbal, A. Arcuri, L. C. Briand, Generating test data from ocl constraints with search techniques, IEEE Transactions on Software Engineering 39 (10) (2013) 1376–1402.

[21] J. Wittevrongel, F. Maurer, 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, 2001, pp. 41–48.

[22] J. Ryser, M. Glinz, A practical approach to validating and testing software systems using scenarios (1999).

[23] W. T. Tsai, A. Saimi, L. Yu, R. Paul, Scenario-based object-oriented testing framework, in: Int. Conf. on Quality Software, IEEE Computer Society, Los Alamitos, CA, USA, 2003, p. 410.

[24] R. V. Binder, Testing object-oriented systems: models, patterns, and tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[25] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, H. Brandl, Momut::uml model-based mutation testing for uml, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–8.

[26] J. Peleska, E. Vorobev, F. Lapschies, Automated test case generation with smt-solving and abstract interpretation, in: M. G. Bobaru, K. Havelund, G. J. Holzmann, R. Joshi (Eds.), NASA Formal Methods - Third International Symposium, NFM 2011, Vol. 6617 of Lecture Notes in Computer Science, Springer, 2011, pp. 298–312.

[27] A. Gargantini, C. Heitmeyer, 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, 1999.

[28] L. Tan, O. Sokolsky, I. Lee, Specification-based testing with linear temporal logic, in: IRI'2004, IEEE Int. Conf. on Information Reuse and Integration, 2004, pp. 413–498.

[29] P. Amman, W. Ding, D. Xu, Using a model checker to test safety properties, in: 7th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'01), IEEE, 2001, p. 212.

[30] P. E. Ammann, P. E. Black, W. Majurski, Using model checking to generate tests from specifications, in: Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241), 1998, pp. 46–54.

[31] G. Fraser, F. Wotawa, Using Model-Checkers to Generate and Analyze Property Relevant Test-Cases, Software Quality Journal 16 (2008) 161–183.

[32] M. Auguston, J. Michael, M.-T. Shing, 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, 2005, pp. 1–6.

[33] G. Carvalho, F. Barros, F. Lapschies, U. Schulze, J. Peleska, Model-based testing from controlled natural language requirements, in: C. Artho, P. C. Ölveczky (Eds.), Formal Techniques for Safety-Critical Systems: Second International Workshop, FTSCS 2013, Queenstown, New Zealand, October 29–30, 2013. Revised Selected Papers, Springer International Publishing, Cham, 2014, pp. 19–35.

[34] V. Rusu, H. Marchand, T. Jéron, Automatic verification and conformance testing for validating safety properties of reactive systems, in: J. Fitzgerald, A. Tarlecki, I. Hayes (Eds.), Formal Methods 2005 (FM05), LNCS, Springer, 2005.

[35] H. Hallal, S. Boroday, A. Ulrich, A. Petrenko, An automata-based approach to property testing in event traces, in: Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems, TestCom'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 180–196.

[36] W.-T. Tsai, L. Yu, F. Zhu, R. Paul, Rapid embedded system testing using verification patterns, IEEE Softw. 22 (4) (2005) 68–75.

[37] B. K. Aichernig, R. Schumi, Property-based testing with fscheck by deriving properties from business rule models, in: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2016, pp. 219–228.

[38] W. Dou, D. Bianculli, L. Briand, Oclr: A more expressive, pattern-based temporal extension of ocl, in: Proceedings of the 10th European Conference on Modelling Foundations and Applications - Volume 8569, Springer-Verlag New York, Inc., New York, NY, USA, 2014, pp. 51–66.