Test Generation and Evaluation from High-Level Properties for Common Criteria Evaluations – The TASCCC Testing Tool –

Frédéric Dadeau*, Kalou Cabrera Castillos*, Yves Ledru†, Taha Triki†, German Vega†, Julien Botella‡ and Safouan Taha§
*FEMTO-ST Institute UMR 6174 - INRIA CASSIS, 16 route de Gray, F-25030 Besançon cedex, France
Email: {frederic.dadeau,kalou.cabrera}@femto-st.fr
†UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble 2/CNRS, LIG UMR 5217, F-38041, Grenoble, France
Email: {yves.ledru,taha.triki,german.vega}@imag.fr
‡Smartesting, 18 rue Alain Savary, F-25000 Besançon, France
Email: julien.botella@smartesting.com
§SUPELEC Systems Sciences - Computer Science Department, 3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex, France
Email: safouan.taha@supelec.com

***Abstract*—In this paper, we present a model-based testing tool resulting from a research project, named TASCCC. This tool is a complete tool chain dedicated to property-based testing in UML/OCL, that integrates various technologies inside a dedicated Eclipse plug-in. The test properties are expressed in a dedicated language based on property patterns. These properties are then used for two purposes. First, they can be employed to evaluate the relevance of a test suite according to specific coverage criteria. Second, it is possible to generate test scenarios that will illustrate or exercise the property. These test scenarios are then unfolded and animated on the Smartesting's CertifyIt model animator, that is used to filter out infeasible sequences. This tool has been used in industrial partnership, aiming at providing an assistance for Common Criteria evaluations, especially by providing test generation reports used to show the link between the test cases and the Common Criteria artefacts.**

***Keywords*-Model-based testing, UML/OCL, property patterns, coverage criteria, test scenarios, test reports, Common Criteria.**

I. Introduction

Starting in 2009, and ending in 2012, the TASCCC project[1] aimed at the assistance of the validation engineer and evaluators in the complex tasks of both preparing and performing a Common Criteria evaluation [1]. The key idea in such evaluations is to be able to provide and exhibit evidence that, among others, the testing phase has been extensively performed. The project thus aimed at two goals. First, it was intended to reduce the test generation effort by offering a new way to automatically generate security-oriented tests. Second, it aimed at simplifying the Common Criteria evaluation process by automatically compiling relevant information to be provided to the evaluator. The project involved three academic partners (FEMTO-ST, LIG and SUPELEC), and four industrial partners (Gemalto, Serma Technologies, Smartesting and Trusted Labs).

To address these two issues, these project partners have proposed two complementary approaches. The first approach aims at using test properties in order to generate model-based test cases. These test properties are expressed using test patterns describing temporal aspects of the system under test. These test properties are then used either for evaluating the relevance of a given test suite, in terms of property coverage, or for generating new model-based test cases. The second approach aims at producing test generation reports that relate the test cases to the various artefacts required by the Common Criteria evaluation. Both approaches have been implemented in a dedicated testing tool framework, called TASCCC, which relies on an existing model-based technology, namely the CertifyIt tool [2] provided by the Smartesting company. This tool takes a UML/OCL model as input and automatically generates functional test cases using a structural coverage criterion, based on the control-flow paths of the OCL code within the operations.

The TASCCC prototype is an Eclipse plug-in that integrates the various technologies provided or developed by the different partners of the project. The workflow of the tool matches the different workpackages of the project summarized in Fig. 1. From the initial requirements, called Functional Security Profile (FSP), the validation engineer extracts test properties that are expressed at the model level. Such properties express the occurrence, absence, or sequences of given events when the system is being executed. The validation engineer then selects the property coverage criterion he wants to apply. The tool automatically computes test scenarios that either illustrate the property, or check the robustness of the system, by attempting to perform forbidden actions. The test scenarios are expressed as regular expressions representing sequences/choices/repetitions of operation calls, possibly reaching specific model states. Such scenarios then need to be unfolded and filtered so as to eliminate sequences that would not be executable at the model level. To do so, two complementary tools interact: TOBIAS [3], which is in charge of unfolding the scenario into (unverified) test cases, and the model animation engine of CertifyIt, which is in charge of animating each test case

[1]funded by the French National Research Agency under grant ANR-09-SEGI-014

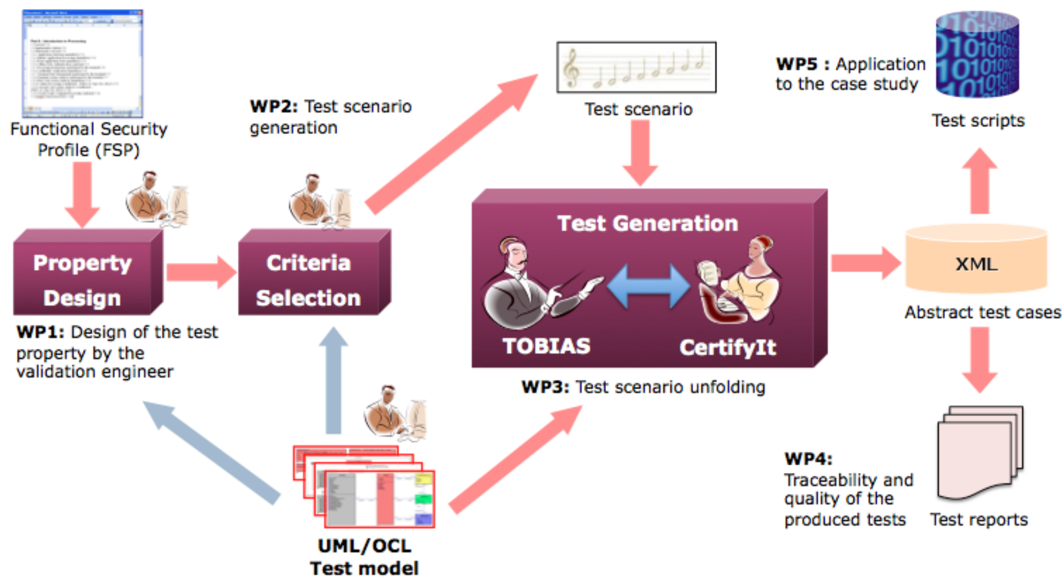

Fig. 1. Overview of the TASCCC process

on the UML/OCL model. If the test case can be entirely animated on the model, it is kept in the CertifyIt test repository. Otherwise, it is discarded. The test repository contains all the test cases produced during the process. It can be exported in a dedicated XML format that provides all the test cases and their characteristics (steps, expected results, covered model artefacts, etc.). This test suite can then be concretized into test scripts to be run on the system under test. In addition, the test suite can be exploited to produce test generation reports that will be used for the Common Criteria evaluations. These reports include traceability information to link test cases with the initial functionalities and properties involved.

This paper describes, and illustrates on the tool, these different steps as follows. Section II briefly explains the UML/OCL models we consider and introduces a running example that will be used to illustrate the various possibilities of the tool. Then, Section III describes the test property description language, explains its semantics and gives the associated property coverage criteria that are implemented within the tool. Section IV shows the two usages of these coverage criteria: the evaluation of an existing test suite, and the generation of test cases, involving the coupling of the TOBIAS tool and the animation engine of CertifyIt. The production of test reports and their relationship with the Common Criteria artefacts is given in Section V. Finally, Section VI summarizes the features of the tool.

II. UML/OCL Models – Running Example

This section introduces the UML/OCL subset that is considered, along with a running example.

A. Considered Subset of UML/OCL

The UML models we consider are those supported by the CertifyIt test generation tool, developed by the Smartesting company. This tool automatically produces model-based tests from a UML model [4] with OCL code describing the behaviors of the operations. CertifyIt does not consider the whole UML notation as input and relies on a subset named UML4MBT which considers class diagrams, to represent the data model, augmented with an imperative form of OCL constraints [5], 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.

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

OCL provides the ability to navigate the model, select collections of objects and manipulate them with universal/existential quantifiers to build first-order logic expressions. Regarding the OCL semantics, UML4MBT does not consider the third logical value *undefined* that is part of the classical OCL semantics. All expressions have to be valuated at run time in order to be evaluated. CertifyIt interprets OCL expressions with a strict semantics, and raises execution errors when encountering null pointers. The overall objective is to take advantage of an executable UML/OCL model. Indeed, the test cases are produced by animating the model according to a given test scenario. Before describing this process, we first introduce our running example.

B. Running Example

The running example we consider represents a web application, called eCinema, that makes it possible, for registered and authenticated users, to buy tickets for the movies played in

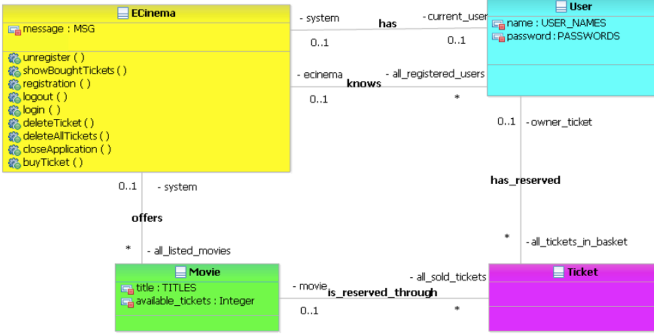

Fig. 2. Class diagram of the eCinema application

the cinema. The UML class diagram is depicted in Fig. 2. The `ECinema` class represents the system under test and provides its API. The `User` class represents the user that can be registered and/or connected to the application. Once connected the user can buy tickets associated to a given `Movie`.

The behavior of the operations is expressed using OCL code. Figure 3 shows the OCL code of the `login` operation that authenticates the user after having checked its login and password. This operation is written in a defensive style, meaning that its precondition is always true. The postcondition thus contains all the possible execution results, that may be correct, or incorrect for any reason.

Notice that code branches are annotated with special tags, starting with `@REQ` or `@AIM` and identifying specific behaviors of the operation. In the CertifyIt test generator, such annotations are used to trace the initial requirements. When a test is computed it is thus possible to know which path in the code has been activated and which of these requirements are covered.

In practice the UML/OCL test model has to be designed using the Rational Software Architect from IBM [6], for which Smartesting provides a plug-in, that makes it possible to export the model in the CertifyIt format. Once the model is designed, it can be used as an input to the TASCCC prototype.

```
context login(in_name,in_password)::effect:
---@REQ: ACCOUNT_MNGT/LOG
if in_name = USER_NAMES::INVALID_USER then
   ---@AIM: LOG_Empty_User_Name
   message= MSG::EMPTY_USERNAME
else
   if not all_registered_users->exists(name=in_name) then
      ---@AIM: LOG_Invalid_User_Name
      message= MSG::UNKNOWN_USER_NAME_PASSWORD
   else
      let user_found : User = all_registered_users
                                    ->any(name = in_name) in
         if user_found.password = in_password then
            ---@AIM: LOG_Success
            self.current_user = user_found and
            message = MSG::WELCOME
         else
            ---@AIM: LOG_Invalid_Password
            message = MSG::WRONG_PASSWORD
         endif
   endif
endif
```

Fig. 3. OCL code of the `login` operation

III. Test Property Design using Patterns

This section describes the test property language, named Temporal OCL [7], that acts as an overlay to the standard OCL constraints to express temporal properties on the system.

A. Syntax and Semantics

The property description language is a temporal extension of OCL. It relies on the proposal of Dwyer *et al.* [8] in which a temporal property is a *temporal pattern* that is applied in a *scope*. Thus, the user can define a temporal property choosing a pattern and a scope among a list of predefined schemas. The scopes are defined from *events* and delimit the impact of the pattern. The patterns are defined from events and *state properties* to characterize execution sequences that are correct. The state properties and the event are described from OCL expressions. This language and its semantics are detailed in [9].

There are two kinds of events. Events denoted by `becomesTrue(`$oclExpr$`)`, where $oclExpr$ is a boolean OCL expression, represent a change in the truth value of a given predicate. This event is satisfied by an operation call when the property $oclExpr$ evaluates to false before the operation call, and to true after the call. The other kind of event is denoted by `isCalled(op, pre, post, {tags})` and represents operation calls. In this expression, `op` designates an operation. `pre` and `post` are OCL predicates respectively representing a precondition and a postcondition. Finally, `tags` represent 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`}.
There are five temporal patterns: (i) `always` $oclExpr$ means that state property $oclExpr$ 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 state in the future. 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 `(directly) precedes` E_2 means that event E_1 (directly) precedes event E_2. (v) E_1 `(directly) follows` E_2 means that event E_2 is (directly) followed by event E_1.

There are five scopes that can apply to a temporal pattern P: (a) P `globally` means that P must be satisfied on any state. (b) P `before` E means that P must be satisfied before the first occurrence of E. (c) P `after` E means that P must be satisfied after the first occurrence of E. (d) P `between` E_1 `and` E_2 means that P must be satisfied between any occurrence of E_1 followed by an occurrence of E_2. (e) P `after` E_1 `unless` E_2 means that P must be satisfied between any occurrence of E_1 followed by an occurrence of E_2 and even after an occurrence of E_1 that is not followed by an occurrence of E_2.

We illustrate this language by formalizing a property of our running example. We want to express an access control policy that states that only authenticated users can successfully buy tickets. We thus express three properties:

Property 1. *"Before logging on the system, it is not possible to buy a ticket"*
never `isCalled(buyTicket(), {@AIM:BUY_Success})`
before `isCalled(login(), {@AIM:LOG_Success})`

Property 2. *"After logging out and before logging in again it is not possible to buy a ticket"*
never `isCalled(buyTicket(), {@AIM:BUY_Success})`
after `isCalled(logout(), {@AIM:LOG_Logout})`
unless `isCalled(login(), {@AIM:LOG_Success})`

Property 3. *"A ticket purchase may happen when the user is connected"*
eventually `isCalled(buyTicket(),{@AIM:BUY_Success})`
at least 0 **times**
between `isCalled(login(), {@AIM:LOG_Success})`
and `isCalled(logout(), {@AIM:LOG_Logout})`

A Temporal OCL editor has been designed during the project[2]. This editor provides classical features such as syntax highlighting and auto-completion based on the elements from the UML model (operation names, class attributes, etc.) and OCL syntax (collection operators, etc.) useful for writing OCL predicates.

B. Property Automata and Coverage Criteria

The semantics of the property pattern language is described as a labelled transition system that expresses the satisfaction of the property. Once the property is loaded, it is automatically translated into an equivalent automaton whose transitions are labelled by events that occur on the system, originating from the textual property.

For example, Figure 4 represents the automaton associated to Property 1 described previously. On this figure, transition E0 represents the `isCalled(login(), {@AIM:LOG_Success})` event, while transition E2 represents the `isCalled(buyTicket(), {@AIM:BUY_Success})` event. These two transitions are called α-transitions, as they are labelled with events that originate from the property. On the opposite, Σ-transitions match the complementary events. This automaton contains one final state (state 2 denoted by a double circle) that is reached once the scope is over (once a successful login has been performed). It also presents an error state (state X) that is reached when the forbidden sequence is performed.

This formalism makes it possible to define coverage criteria. Intuitively, one can guess that classical coverage criteria (all-nodes, all-edges, all-k-paths) are not relevant here, since

[2]A standalone editor is also available from the author's personal web page: http://wwwdi.supelec.fr/taha/temporalocl/

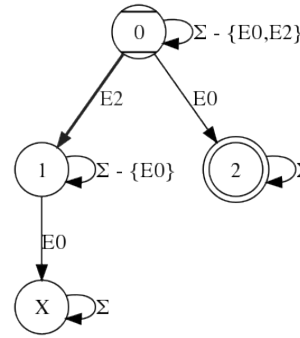

Fig. 4. Property 1 Automaton

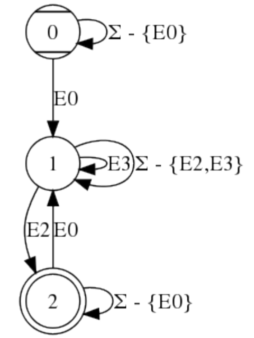

Fig. 5. Property 3 Automaton

(i) all nodes are not necessarily reachable (if we assume that the model satisfies the property, then the error states are unreachable), (ii) all transitions are not of equal significance (Σ-transitions are not really interesting, contrary to α-transitions, as they only represent internal actions of the system that are used to activate α-transitions), and (iii) k-path coverage should make a distinction between the loops in the scope/pattern part of the automaton and the reflexive Σ-transitions. We thus introduce dedicated coverage criteria.

1) Nominal Coverage Criteria: We now define the 4 nominal coverage criteria that we consider, adapted from classical automata coverage criteria. We illustrate these criteria on the automaton depicted in Fig. 5 which represents Property 3. In this figure, event E0 represents a successful login, event E2 represents a logout, and E3 represents a successful ticket purchase.

α-transitions coverage consists in covering the α-transitions of the property automaton. On the example, this coverage makes it possible to produce/identify test cases that perform the following sequence E0 . . . E3 . . . E2.

α-transition-pairs coverage consists in covering relevant pairs of α-transitions on the automaton. The considered pairs depend on the considered scope/pattern combination. On the example, relevant α-transition-pairs will be (E0,E3) and (E3,E2).

k-pattern coverage consists in performing k iterations of the part of the automaton that corresponds to the property pattern. On the example, this consists in iterating the E3 transiton without exiting the surrounding scope. Such a criterion may not be applicable if the pattern is not repeatable (e.g. never or eventually . . . at most).

k-scope coverage consists in performing k iterations consisting in entering and exiting the scope of the property and activating an α-transition of the pattern when possible. On the example, this coverage makes it possible to cover path containing E0 (. . . E2 . . . E0 . . .)* E2.

2) Robustness Coverage Criterion: The four previous coverage criteria make it possible to illustrate a given property. However, their relevance for some combination of scope/pattern is arguable. Indeed, if we consider Property 1, whose automaton is given in Fig. 4, that states the absence of a given

event, it is not possible to illustrate such property using the above-mentioned criteria. The only applicable criterion is the α-transition coverage that will only target transition $0 \stackrel{E0}{\rightarrow} 2$ (the other two transitions are not considered since they lead to an error state). Nevertheless, it would be relevant to check that it is not possible to successfully buy a ticket before logging on the system, by forcing the purchase (without expecting it to succeed), and terminating with a successful login.

In order to highlight such test sequences on the automaton, we have defined a robustness criterion that acts as follows. First, the automaton is modified so that the error state becomes the only final state of the automaton. Second, the event labelling a transition leading to this state is weakened, so as to make it enabled on the model. In order to weaken the event, the tool automatically performs some mutations on the event, which can be classified into two categories:

- Modification/Deletion of the predicates in the pre/post states of the event
- Modification/Deletion of the tags described in the event.

On our example of Property 1, we apply a tag deletion mutation, which rewrites `isCalled(buyTicket(), {@AIM:BUY_Success}` to `isCalled(buyTicket())`.

Then, the robustness criterion requires a test to cover the mutated transition and then reach the final state of the automaton to be satisfied.

These coverage criteria can be used at two purposes: evaluating the coverage of an existing test suite, or generating test cases. These two features are now described.

IV. Exploiting the Test Coverage Criteria

We present in this section how the previously described coverage criteria are used in the TASCCC prototype.

A. Evaluation of an existing Test Suite

The first idea is to exploit these coverage criteria to check that an existing test suite satisfies them, fully or partly. The tool proceeds as follows. The test cases are replayed on the UML/OCL model, using the CertifyIt animation engine, while the exploration of the property automaton is done in parallel. Each event that is matched activates a corresponding transition in the automaton. In some cases, it is possible that the exploration of the automaton reaches the error state, meaning that the property is violated. In this case, the error can be in the model, meaning that it does not respect the property, but it may also be the property that is incorrectly specified. Once the test suite has been fully executed, the tool generates a test coverage report.

Figure 6 shows the content of a web page produced to evaluate the nominal coverage of a given property by a test suite. This page contains global statistics about the coverage of the property, and then provides a detailed view of the property coverage for each test case. Green lines indicate that the test case reached a final state of the automaton, while grey lines indicate that the test case was inconclusive (w.r.t. the property),

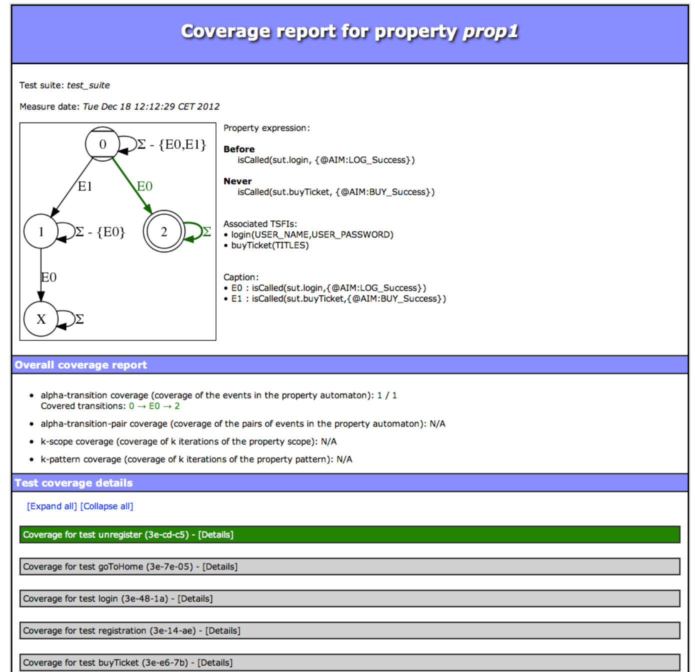

Fig. 6. Nominal Property Coverage Measure Report

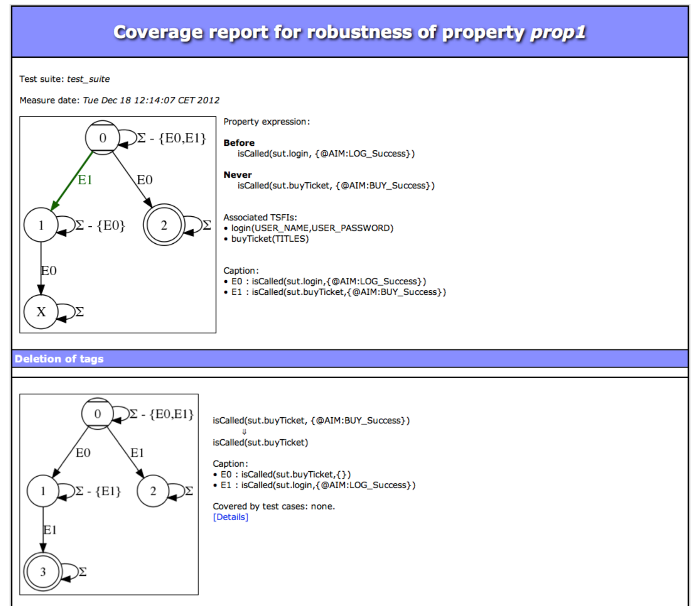

Fig. 7. Robustness Property Coverage Measure Report

i.e. that it ended without reaching a final or error state. Finally, red lines indicate a detected violation of the property.

When performing a robustness coverage measure, a similar web page is obtained, as shown in Fig. 7. This latter presents and summarizes, for the considered property, the applicable mutations and the resulting modified automata, along with a summary of their coverage by the test suite. For each modified automaton, a web page similar to the one given in Fig. 6, is also generated.

B. Test Scenario Generation and Unfolding

When the coverage of a property by the existing test suite is not satisfactory, the validation engineer may produce

TASCCC Tests TSFIs

TSFI	Actions	SFRs	Tags	Tests
logout				
buyticket 100%	Check that the user is logged on the system	FIA_ACC.1	AIM : BUY_Login_Mandatory	buyTicket (3e-e6-7b)
			AIM : BUY_Success	deleteTicket (3e-56-27) buyTicket (3e-7b-18)
	Successfully buy a ticket		AIM : BUY_Success	deleteTicket (3e-56-27) buyTicket (3e-7b-18)
login				

TASCCC © copyright

Fig. 8. Test Generation Report

additional test cases to increase the coverage objective he wants to achieve.

In that sense, the tool can generate dedicated test scenarios that, once unfolded and instantiated on the UML model, will produce abstract test cases satisfying the considered coverage criterion. The generated test scenarios are expressed in TSLT, the Test Schema Language for TOBIAS. TSLT proposes a human-readable high-level format in which the validation engineer can write test schemas. These schema describe sequences of operations, choices between sequences, iterations of sequences, along with state predicates describing intermediate states that must be reached by the test cases once unfolded and executed.

On Property 1, the test scenario generator, for robustness test cases, will produce a scenario such as

$$(\Sigma - \{E0, E2'\}) * \ . \ E2' \ . \ (\Sigma - \{E0\}) * . \ E0$$

in which $E2'$ represents the mutated $E2$ event. In practice, the $(\Sigma - \{\ldots\})*$ expressions will be replaced by a choice between the different operations, representing the considered events, except those appearing in the restriction. In addition, the $*$ iteration operator has to be instantiated to a reasonable value. Once exported in TSLT, all the events are instantiated to operation calls.

The TOBIAS combinatorial tool is then used to unfold such a regular expression. The resulting test sequences are then played on the model, using the CertifyIt animation engine. If a test sequence is not feasible, the test case is discarded. Infeasible sequences may arise for different reasons at some point of the scenario execution. First, it is possible that the expected operation precondition is not satisfied. Second, the requested operation behavior may not be activated. Third, the operation may result in a state that does not fulfil the expected postcondition. Fourth, and finally, the expected state may not be reached. Unfolding a scenario can lead to a huge number of candidate test cases. For example, a robustness scenario created from Property 1 counts 1.89 billion test cases and it would take too much time to animate every test case. Therefore, an incremental evaluation algorithm has been used to discard all test cases corresponding to the same prefix as soon as this prefix fails [3]. Using this incremental evaluation, the tool can select the 126 valid test cases out of 1.89 billion candidates, in less than 4 minutes on a standard PC.

If a test sequence is feasible, meaning that it can be animated entirely, then it is kept in the CertifyIt test repository, enriching the existing test suite. The test suite can then be exported into a specific XML test file that contains all the information regarding the test cases (steps, expected output results, covered requirements, etc.).

V. Test Report Generation

The last feature of the TASCCC tool aims at helping the validation engineers (resp. the Common Criteria evaluators) to prepare (resp. perform) an evaluation of their product. According to the Common Criteria standard [1], the validation engineer is requested to associate to each test case the following artefacts:

- the *TSFI* (TOE –Target Of Evaluation– Security Functional Interface) namely the operations of the system containing sequences of actions (e.g. the `buyTicket` command can be seen as a TSFI).
- the *actions* of a TSFI representing informal steps of the operation execution (e.g. for the `buyTicket` TSFI, we may have an action specified as “the application shall check the authentication of the user”).
- the *SFR* (Security Functional Requirements) associated to each action. They specify high level security functions that may be provided by the system. (e.g. the previous action can be associated to an access control SFR, such as FDP_ACC.1 “the TSF shall allow the buyTicket operation for a authenticated user”).

In the current practices, the link between these elements is provided in the design document, meaning that the validation engineer asserts that the test covers such SFR, but no further evidence is provided. The TASCCC tool proposes to exploit additional informations to draw a strong and indisputable relationship between the tests and these artefacts.

The connection with the model is made through the actions, that are associated to @AIM and @REQ tags in the OCL code of the operations. On the example, the action describing the verification of the user’s authentication is ensured by two sets of tags {@AIM:Login_First_Mandatory}, which

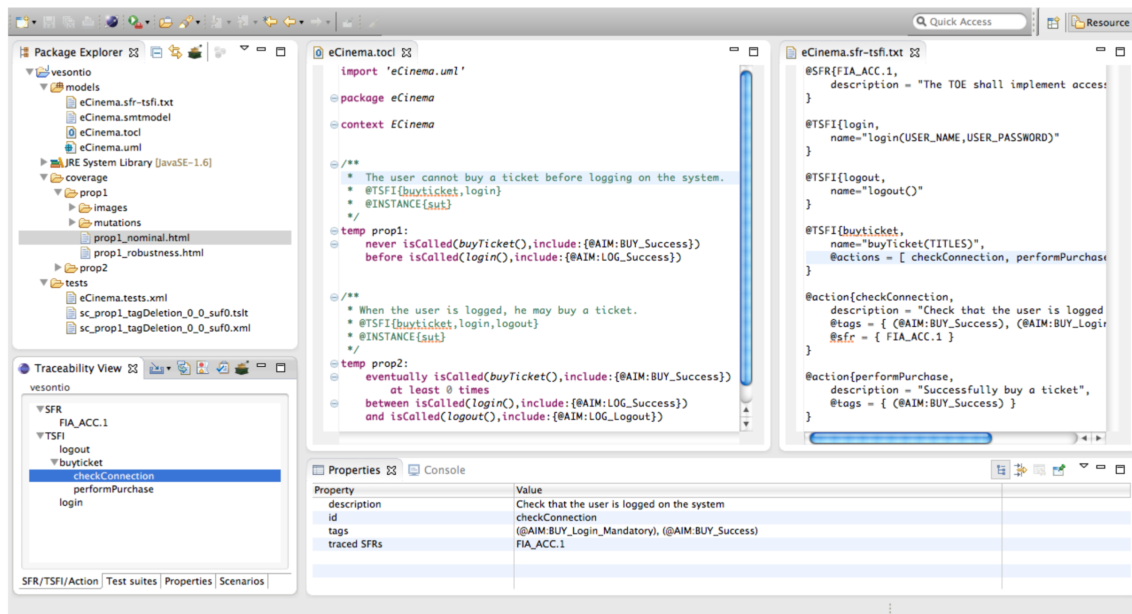

Fig. 9. Main Frame of the TASCCC Prototype

captures the error case when the user is not registered, and {@AIM:BUY_Success} which captures the nominal case, happening when the user is actually authenticated. In practice, the definition of the TSFI/actions/SFR is made in a text file, using a simple key–value format, that is used to describe these artefacts, and their relationships.

By exploiting this information and the test suite repository, the TASCCC tool makes it possible to automatically create a test generation report, that provides all elements of information requested by the Common Criteria evaluator. Figure 8 gives an overview of the information available in the Common Criteria report. The web page presents, for each TSFI, the set of associated actions, and, for each action, the SFR to which they relate. Finally, test cases are displayed in front of each action they cover.

VI. Summary of the TASCCC Testing Tool

This paper has presented the tool prototype resulting from the TASCCC project, funded by the French National Research Agency. This prototype integrates, in a dedicated Eclipse plug-in, various technologies for property-based testing, relying on a UML/OCL model. The main frame of the tool is represented in Figure 9. The editor view (center of the frame) makes it possible to edit the various files involved in the process: (i) the TSFI/actions/SFR description, (ii) the TOCL properties, and (iii) the TSLT scenarios.

The plug-in provides a dedicated view, called *Traceability view* (bottom-left corner of the frame) which displays the artefacts that are associated to a given project, namely the description of the TSFI/actions/SFR, the existing test suites, the properties defined by the validation engineer, and the existing test scenarios. This view provides the main commands that make it possible to:

- measure the coverage of a given test suite. When executed, the wizard asks the user to choose the property he wants to measure and the kind of coverage criteria that have to be considered (nominal or robustness). Finally, the user is asked to provide the directory in which the coverage measure report will be generated.
- produce a test scenario from a given property. The wizard asks the user the coverage criterion he wants to apply to generate the test scenario. When produced, the test scenario keeps track of the originating test property and the coverage criteria that was applied.
- generate the TSLT file for a given scenario. This file may then be edited, before running TOBIAS coupled with the CertifyIt animator to unfold the scenario and instantiate the test cases.
- create a test generation report for a given test suite. The wizard requires the user to provide the output directory. By taking into account the TSFI/actions/SFR mapping associated to the project, the tool produces the test generation report.

It is important to notice that the two approaches that are implemented within the TASCCC tool can be used independently. This prototype has been applied on an industrial case study of GlobalPlatform, a next-generation operating system for smart cards, provided by Gemalto. The first feedbacks from both the validation engineers of Gemalto and the Common Criteria evaluators are very positive on the benefits of the tool w.r.t. their validation/evaluation activities.

The TASCCC tool differs from existing UML/OCL-based testing tools. For example, HOL-TestGen [10] proposes to decompose the OCL code using DNF to produce the test targets. The approach proposed in [11] relies on a

search-based solver for OCL expressions used to generate object test data. Similarly, the tool proposed in [12] combines mutation and constraint solving. These tools mainly focus on building test data from object oriented specifications in OCL. However, none of them takes into account the dynamics of the system, as done in our approach.

Finally, there exists a couple of limitations of the tool. First, the property language that we have defined is an extension of Dwyer's property patterns. As such, in the original paper, the authors have shown that such a language was able to capture 92% of the properties one may want to express on a system. However, the language does not make it possible to describe scopes or patterns that may involve sequences of events (e.g. A follows B follows C). To overcome this issue, it could be possible to extend the property language syntax. Such an extension would not be problematic since the language constructs are used to derive an automaton from which the rest of the treatment is realized. A second limitation concerns the scalability of the tool. Even though it is possible to handle and animate very large models (several dozens of megabytes), there is still a combinatorial explosion induced by the scenarios themselves, even if smart filtering techniques are implemented in the tool. To overcome this issue, the user still has the possibility to instantiate some or all parts of the scenario (e.g. by reducing the domain of the operation parameters or by replacing unspecified operation calls) by editing the TSLT code produced by the tool. In practice, these two limitations did not prevent the application of the tool on an the industrial case study used in the TASCCC project.

Please visit https://vimeo.com/53210102 to watch a demo.

References

[1] "Common Criteria for Information Technology Security Evaluation, version 3.1," CCRA, Tech. Rep. CCMB-2009-07-001, july 2009.

[2] "Smartesting CertifyIt Test Generator," http://www.smartesting.com/.

[3] T. Triki, Y. Ledru, L. du Bousquet, F. Dadeau, and J. Botella, "Model-based filtering of combinatorial test suites," in *FASE'2012, 15th Int. Conf. on Fundamental Approaches to Software Engineering*, ser. LNCS, J. de Lara and A. Zisman, Eds., vol. 7212, Tallinn, Estonia, Mar. 2012, pp. 439–454.

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

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

[6] D. Leroux, M. Nally, and K. Hussey, "Rational Software Architect: A tool for domain-specific modeling," *IBM Systems Journal*, vol. 45, no. 3, pp. 555–568, 2006.

[7] B. Kanso and S. Taha, "Temporal Constraint Support for OCL," in *Proceedings the 5th International Conference on Software Language Engineering - SLE2012*, ser. LNCS 7745, K. Czarnecki and G. H. (Eds.), Eds. Dresden, Allemagne: Springer, 2013, pp. 83–103.

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

[9] K. Cabrera Castillos, F. Dadeau, J. Julliand, and S. Taha, "Measuring test properties coverage for evaluating UML/OCL model-based tests," in *ICTSS'11, 23-th IFIP Int. Conf. on Testing Software and Systems*, ser. LNCS, B. Wolff and F. Zaidi, Eds., vol. 7019. Paris, France: Springer, Nov. 2011, pp. 32–47.

[10] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A specification-based test case generation method for UML/OCL" in *Proceedings of the 2010 international conference on Models in software engineering*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 334–348.

[11] S. Ali, M. Iqbal, A. Arcuri, and L. Briand, "A search-based OCL constraint solver for model-based test data generation," in *Quality Software (QSIC), 2011 11th International Conference on*, july 2011, pp. 41 –50.

[12] B. Aichernig and P. Salas, "Test case generation by OCL mutation and constraint solving," in *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, sept. 2005, pp. 64 – 71.