j-POST: a Java Toolchain for Property-Oriented Software Testing

Yliès Falcone[1], Laurent Mounier[1], Jean-Claude Fernandez[1], and Jean-Luc Richier[2]
`Firstname.Lastname@imag.fr`

[1] Verimag, Université Grenoble I, INPG, CNRS
[2] LIG Laboratory, Université Grenoble I, INPG, CNRS

Abstract. j-POST is an integrated toolchain for property-oriented software testing. This toolchain includes a test designer, a test generator, and a test execution engine. The test generation is based on an original approach which consists of deriving a set of *communicating test processes* obtained both from a requirement formula (expressed in a trace-based logic) and a behavioral specification of some specific parts of the software under test. The test execution engine is then able to coordinate the execution of these test processes against a distributed Java program. j-POST was applied to check the correct deployment of a security policy for a travel management application.

1 Introduction

Testing is a validation technique aimed to find defective behaviours on a system either during its development, or once a final version is issued. It remains one of the most feasible methodologies to ensure the expected behaviour of a software. This is notably due to its ability to cope with continual growth of system complexity. However, reducing its cost and time consumption remains a very important challenge sustained by a strong industrial demand.

In previous work [1, 2] we have presented a black-box test generation method able to construct abstract test cases from a formal requirement (a property that the system is expected to fulfill). This method (implemented in a prototype tool) is based on a test calculus allowing the method to be compositionally and formally defined. In this framework, a requirement is expressed by a logical formula built upon a set of (abstract) predicates. Each predicate corresponds to a (possibly non-atomic) operation to be performed on the system under test, and is user-provided as a *test module* indicating how to perform this operation on the actual implementation, and how to decide whether its execution succeeds or not. The test generation step consists in building, by *composition* of test modules, a set of *communicating test processes* from this property. In this paper we present a significant step from this previous work. First off, we present formally how the previously generated test can be executed. Besides we present j-POST, an integrated toolchain for property-oriented software testing. In addition to a full implementation of the test generation tool, we present the associated test designer and test execution engine resulting in a fully integrated toolchain. The test designer helps the user to provide inputs to the test generator. The test execution engine is able to coordinate the execution of the generated processes against a possibly distributed program, leading to a satisfiability verdict with respect to the given requirement.

Comparison with classical model-based testing. This approach offers several advantages over more classical model-based test generation techniques [3] implemented in several existing tools (*e.g.* TGV [4], TorX [5], see [6, 7] for more exhaustive surveys). First, j-POST is able to deal with *piecewise specifications* restricted to specific functionalities. We strongly believe that this feature is really important in practice, especially in application domains where formal modeling of software is not a common practice. Specifying only some global requirement and some specific implementation features in an operational way (*i.e.* the test modules) seems much easier for test engineers than building a complete model of a software. As a consequence, the test generation step will not require the exploration of such a complete model, avoiding the well-known state explosion problem. Furthermore, this toolchain remains *open* in the sense that various logics can be considered to express the requirements, and new logic plugins can be easily added. Finally, this toolchain integrates a large spectrum of the whole test process, from the test design to the test execution.

The remainder of this paper is organized as follows. Sect. 2 describes the underlying theory of j-POST and Sect. 3 describes the toolchain itself. In Sect. 4, we depict one of the experiments conducted with j-POST on a travel agency application. Sect. 5 exposes some conclusions and perspectives opened by this work.

2 Underlying testing theory

This section briefly presents the background of j-POST, namely how to produce and execute test cases from a formal requirement following a syntax-driven approach. More details can be found in the research reports available in [8].

We consider in the following that the behaviour of the software under test (SUT) can be modelled using a *labelled transition system* (LTS), noted Sut, namely a quadruplet $(Q^{Sut}, Act^{Sut}, \rightarrow, q^0)$ where Q is a set of states, Act^{Sut} a set of actions (labels), $\rightarrow \subseteq Q^{Sut} \times Act^{Sut} \times Q^{Sut}$ the transition relation and $q^0 \in Q^{Sut}$ the initial state. In *black-box* testing this behaviour can be accessed only through a SUT interface, namely a set of *visible* actions $Act^{vis} \subseteq Act^{Sut}$. Non visible actions are supposed to be labelled by τ. We will denote by $p \xrightarrow{a} q$ when $(p, a, q) \in \rightarrow$, and by $p_1 \overset{\tau^* a}{\Longrightarrow} q$ when there exist $p_2, p_3, \ldots, p_n$ s.t. $p_i \xrightarrow{\tau} p_{i+1}$ and $p_n \xrightarrow{a} q$. Finally, we define the *execution sequences* of Sut as the set of *finite* sequences of visible actions that can be performed from its initial state: $Exec(Sut) = \{a_1.a_2.\cdots.a_n \mid \exists q_1, \ldots, q_{n+1}\ s.t.\ q_i \overset{\tau^* a_i}{\Longrightarrow} q_{i+1} \wedge q_1 = q^0\}$.

2.1 The properties to test

We assume in the following that the properties to test are expressed using a logic $\mathscr{L}$. Formulas of $\mathscr{L}$ are built upon a finite set of *n-ary operators* F^n and a finite set of *predicates* $\{p_1, p_2, \ldots, p_n\}$. The abstract syntax of such a logic could be defined as follows: formula ::= $F^n(\text{formula}_1, \text{formula}_2, \ldots, \text{formula}_n) \mid p_i$.

Formulas of $\mathscr{L}$ are interpreted over *finite* execution sequences. However, this semantics also takes into account two other important features:

– First, this semantics is defined on two levels. Predicates are not *atomic*, *i.e.* they do not necessarily correspond to occurrences of *single* visible actions, but rather of (concrete) *sequences* of visible actions. Operators F^n are then interpreted over *abstract* execution sequences, *i.e.*, sequences of predicates.
– Second, since our objective is to *test* either the validity or the non-validity of a formula φ, the semantics of φ defines three kinds of execution sequences, corresponding to the possible verdicts delivered by a tester: the ones that *satisfy* φ (pass), the ones that *do not satisfy* φ (fail), and the ones for which *we cannot conclude* about the satisfiability of φ (inconc).

More formally, a triplet of finite languages $(L^P_{p_i}, L^F_{p_i}, L^I_{p_i})$ is associated with each predicate p_i. These three languages define respectively concrete execution sequences that satisfy p_i, that do not satisfy p_i, and for which the satisfiability of p_i is unknown. The following assumptions are required:

– $L^P_{p_i}$, $L^F_{p_i}$ and $L^I_{p_i}$ are defined over an alphabet $A_{p_i} \subseteq Act^{vis}$. Intuitively, A_{p_i} is the set of visible actions whose occurrences influence the truth value of p_i.
– This set of three languages defines a *partition* of $(A_{p_i})^*$.
– For two distinct predicates p_i, p_j, A_{p_i} and A_{p_j} are disjoint.

The semantics of a non-atomic formula $\varphi(p_1, p_2, \ldots, p_n)$ is then defined by three sets $[\![\varphi]\!]^P$, $[\![\varphi]\!]^F$, $[\![\varphi]\!]^I$, inductively computed from $L^P_{p_i}$, $L^F_{p_i}$ and $L^I_{p_i}$ for each p_i appearing in φ.

Finally, we say that an LTS S satisfies φ (we note $S \models \varphi$) iff *all* sequences of $Exec(S)$ belong to $[\![\varphi]\!]^P$, and we say that it does not satisfy φ iff there exists a sequence of $Exec(S)$ that belongs to $[\![\varphi]\!]^F$.

2.2 A set of communicating test processes

The test cases we aim to produce consist of a set of *sequential communicating test processes*. Roughly speaking, each test process is built from classical programming primitives such as variable assignment, sequential composition, (non-deterministic) choice, and iteration. It can also perform communications with the other test processes, and interact with the SUT. This sequential behaviour can be modelled by an LTS extended with variables.

Test processes run asynchronously and communicate with each other either by "rendez-vous" on dedicated communication channels or through shared variables. The semantics of a whole test process T_φ can be expressed by an LTS S_{T_φ}. A complete syntax and semantics of such a "test calculus" can be found in [1], but other classical process algebra could be used as well.

2.3 Test generation

The purpose of the test generation phase [2] is to produce a test case T_φ (*i.e.* a set of communicating test processes) associated to the $\mathscr{L}$-formula φ under test. We distinguish two kinds of test processes (which are both LTSs):

- *test modules* t_{p_i}, provided by the user, and associated with the predicates p_i of φ. Their purpose is to produce a test verdict indicating whether a given concrete execution sequence belongs either to $L^P_{p_i}$, $L^F_{p_i}$ or $L^I_{p_i}$. Examples of such test modules are given on Fig. 5 in Sect. 4.
- *test controllers* t_{F^n}, associated with each n-ary operator F^n of the logic $\mathscr{L}$. Their purpose is to control the execution of the test process associated to each of their operands by means of basic signals (start, stop, loop), and to collect their verdicts in order to produce a resulting verdict corresponding to this instance of operator F^n. One can find controllers for several logics in the research reports provided in [8].

This test generation technique can be formalized by a function called $\mathit{GenTest}_{\mathscr{L}}$, such that $\mathit{GenTest}_{\mathscr{L}}(\varphi) = T_\varphi$. This function is inductively defined on the syntax of $\mathscr{L}$ in the following way:

- If $\varphi = p_i$, then $\mathit{GenTest}_{\mathscr{L}}(\varphi)$ returns the test module t_{p_i} (associated with the predicate p_i) extended with the communication operations required to make it controlable by another test process (see Fig. 3 in Sect. 3).
- If $\varphi = F^n(\varphi_1, \cdots, \varphi_n)$, then $\mathit{GenTest}_{\mathscr{L}}(\varphi)$ returns a parallel composition between (recursively defined) test processes $t_{\varphi_1}, \ldots, t_{\varphi_n}$ and an instance of the (generic) test controller t_{F^n}.

Finally, a special test process t_{main} is added to launch the whole test execution and collect the final verdict. According to this generation technique, the architecture of a test case T_φ exactly matches the abstract syntax tree corresponding to formula φ: the root is t_{main}, leaves are test modules corresponding to predicates p_i of φ, and intermediate nodes are controllers associated with operators of φ (see Fig. 6 in Sect. 4 for an example).

2.4 Test selection and execution

From a formal point of view, the test execution sequences are the execution sequences of a parallel composition between the LTS S modelling the SUT behaviour and the test case S_{T_φ}, with a "rendez-vous" synchronization on the visible actions appearing in S_{T_φ}.

However, this LTS product may still contain a bunch of possible test executions (due to possible non-determinism both inside the test modules and introduced by the parallel composition). Moreover, the test generation function only ensures that the verdicts produced by the test execution are *sound* with respect to the initial formula φ: it does not help to select the *interesting* test executions that are likely to exhibit an incorrect behaviour of the SUT. To solve this problem we propose to use *behavioural test objectives*, already introduced in several model-based testing tools (*e.g.* in [4, 9]). Their purpose is to inject some execution scenario in the test cases produced by the test generation phase, either by enforcing the execution order of some visible actions, or by introducing other additional visible actions to lead the SUT into some particular state. Most of the time, in specification based testing, this test selection is performed during the test generation phase, by pruning the undesired test executions from the whole SUT specification. In our approach, the selection is not performed during test generation, but during the test execution (similarly to walk guidance in TorX). This is due to the fact that we do not rely on such a specification. So, the test selection phase is combined with the test execution: the test objective is expressed by an LTS with *accepting* states, and the test sequences leading to such states are privileged during the text execution.

This approach is formalized below. In a LTS S, for two states q, q' we note $q' \in \mathit{Reach}_S(q)$ the fact that q' is accessible from q in S. Also, when $q' \in \mathit{Reach}_S(q)$, we note $d_S(q, q')$ the distance between q and q', *i.e.* the minimal length of the existing paths between q and q'.

Definition 1 (Behavioural test objective). *A test objective O relatively to a test case t which semantics can be expressed by a LTS $(Q^{S_t}, \mathit{Act}^{S_t}, \rightarrow_{S_t}, q_0^{S_t})$ is a deterministic LTS $(Q^O, \mathit{Act}^O, \rightarrow_O, q_0^O)$ complete wrt. Act^{S_t} (i.e. $\forall q \in Q^{S_t}, \forall a \in \mathit{Act}^{S_t}, \exists q' \in Q^O \cdot q \xrightarrow{a}_O q'$). Q^O contains two sink states Accept^O and Reject^O, and $\mathit{Act}^O \subseteq \mathit{Act}^{vis}$.*

Using a test objective, it is possible to operate "on the fly" a test selection on the test case during the execution.

Definition 2 (Selection using a behavioural test objective). *Let t be a test case which semantics can be expressed by $S_t = (Q^{S_t}, \mathit{Act}^{S_t}, \rightarrow_{S_t}, q_0^{S_t})$, and a behavioural test objective $O = (Q^O, \mathit{Act}^O, \rightarrow_O, q_0^O)$. The execution of t guided by O can be defined as a synchronous product $S_t^O = S_t \times O$ such as $\mathit{Act}^{S_t^O} = \mathit{Act}^O$, $Q^{S_t^O} \subseteq Q^{S_t} \times Q^O \cup \mathit{Inc}$, and $\rightarrow_{S_t^O}$ is defined by the following rules. Note that control and observation actions are not distinguished.*

$$\frac{t \xrightarrow{a}_{S_t} t' \qquad \mathit{Accept}^O \in \mathit{Reach}_O(o), o \xrightarrow{a}_O o' \qquad d_S(o', \mathit{Accept}^O) < d_S(o, \mathit{Accept}^O), a \in \mathit{Act}^{S_t}}{(t, o) \xrightarrow{a}_{S_t^O} (t', o')} \quad (1)$$

$$\frac{\mathit{Accept}^O \in \mathit{Reach}_O(o), o \xrightarrow{a}_{S_O} o' \qquad d_S(o', \mathit{Accept}^O) < d_S(o, \mathit{Accept}^O), a \notin \mathit{Act}^{S_t}}{(t, o) \xrightarrow{a}_{S_t^O} (t, o')} \quad (1')$$

$$\frac{t \xrightarrow{a}_{S_t} t' \qquad \mathit{Accept}^O \in \mathit{Reach}_O(o), o \xrightarrow{a}_O o' \qquad d_S(o', \mathit{Accept}^O) \geq d_S(o, \mathit{Accept}^O)}{(t, o) \xrightarrow{a}_{S_t^O} (t', o')} \quad (2)$$

$$\frac{o \xrightarrow{a}_O \mathit{Reject}^O}{(t, o) \xrightarrow{a}_{S_t^O} \mathit{Inc}} \quad (3)$$

Some priorities are associated with these rules to favour the execution of transitions bringing closer to an *Accept* state. The rules (1) and (2) are of the highest prioriy, then is rule (2), and at last the rule (3) is of the lowest priority.

Finally, the set of test execution sequences obtained from an SUT S and a test case S_{T_φ} when taking into account a test objective O is defined as the execution sequences of the parallel composition between the SUT S and the LTS $S_{T_\varphi} \times O$. Note that when the Inc state is reached in this composition, the whole test execution is stopped and an inconclusive verdict is issued. This general framework has been instantiated for two particular logics, namely LTL-X, and extended regular expressions (see Sect. 3.2).

3 Architecture and functionalities of j-POST

The architecture of the toolchain is depicted in Fig. 1. It is built upon three main components, a test designer, a test generator and a test execution engine. Two interfaces are provided: a command-line mode and a graphic interface.

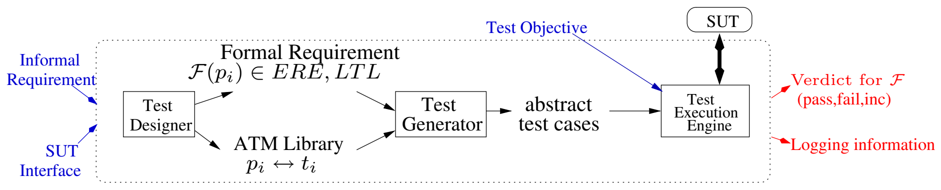

Fig. 1. Abstract view of the j-POST testing toolchain

The purpose of j-POST is to check through black-box testing whether a Java application fulfills a given requirement. To do so, the *test designer* (step 1) helps the user both to formalize this requirement in a trace-based logic and to elaborate a test module library. Each test module (corresponding to a predicate used in the requirement) is obtained by combining some of the actions offered by the SUT interface. The test modules are used by the *test generator* (step 2), according to a logic plugin, to produce a test case as a set of communicating test processes. Finally, this test case can be launched by the *test engine* (step 3), taking into account a test objective to select the more promising test sequences.

3.1 Test designer

The test designer of j-POST is a user assistant that helps to elaborate the formal requirements and the corresponding test modules through dedicated editors available within the Eclipse Modeling Framework. Each test module is stored into an XML file (their j-POST internal representation). Moreover, the test designer provides a tool (based on GraphViz [10]) to vizualise them in a more intelligible way. This avoids any error-prone manipulations of XML files from the user.

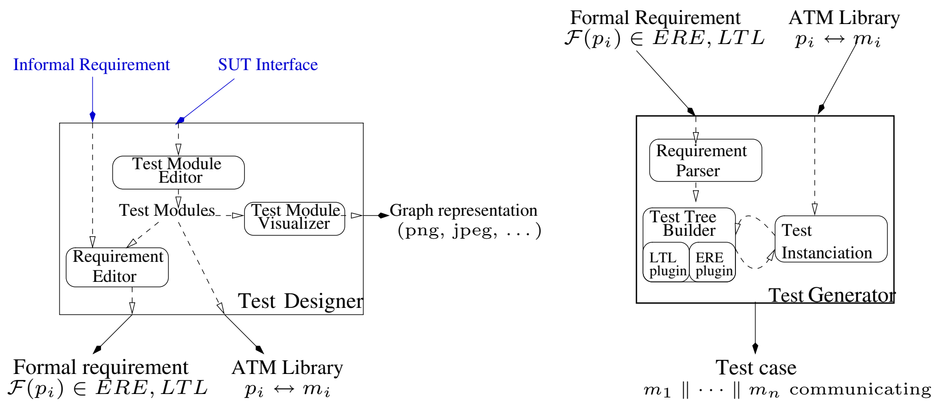
Informal Requirement
SUT Interface
Test Module Editor
Test Modules
Test Module Visualizer
Graph representation (png, jpeg, . . .)
Requirement Editor
Test Designer
Formal requirement $\mathcal{F}(p_i) \in ERE, LTL$
ATM Library $p_i \leftrightarrow m_i$
Formal Requirement $\mathcal{F}(p_i) \in ERE, LTL$
ATM Library $p_i \leftrightarrow m_i$
Requirement Parser
Test Tree Builder
LTL plugin
ERE plugin
Test Instanciation
Test Generator
Test case $m_1 \parallel \cdots \parallel m_n$ communicating

Fig. 2. Abstract view of the j-POST test designer and test generator architecture

3.2 Test generator

The j-POST test generator consists mainly in implementing the $GenTest_{\mathscr{L}}$ function. It produces a test case following the syntax-driven approach recalled in Sect. 2.3 in two stages:

The first stage is the construction of a communication tree obtained from the abstract syntax tree of the formula. This tree expresses the communication architecture between the test processes that will be produced by the test generator. Its leaves are abstract test modules (ATM) corresponding to the atomic predicates of the formula, taken from the library. Its internal nodes are (copies of) generic test controllers, corresponding to the logical operators appearing in the formula (they are obtained from a finite set of generic controllers provided by the logic plugin). Finally, the root of this tree is a special test process, called `testCaseLauncher`, whose purpose is to initiate the test execution and deliver the resulting verdict.

The second stage consists of *instantiating* the communication tree by associating fresh channel names to each local communication between test processes. It relies on a traversal of this communication tree in order to modify the test modules. In particular the test modules provided by the user are automatically extended with additional communication actions to be managed by the test controllers, *e.g.* starting signal, verdict emission (see Fig. 3). The resulting test case is a set of XML files, one per test process.

Generic test controllers and test generation algorithms have been defined for different specification formalisms. So far, j-POST TestGenerator supports two common-use formalisms, by means of *logic plugins*:

– Temporal logics [11] like LTL are frequently used in the verification community to express requirements on reactive systems. We consider here fragments of such logics whose models are set of *finite* execution traces. We did not include the next operator in order to be insensitive to stuttering [12]. The complete definition of the variant of LTL-X we use is given in [2].

– Extended Regular Expressions [13] are another formalism to define behavior patterns expressed by finite execution traces. They are commonly used and well-understood by engineers.

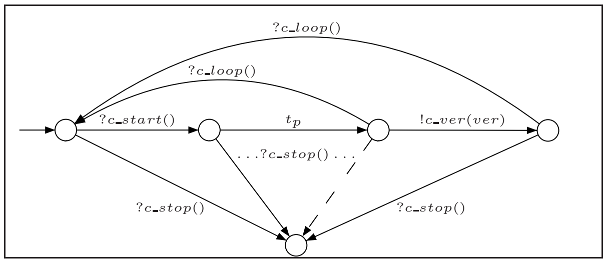

Fig. 3. Instantiation of an abstract test component t_p

3.3 Test execution engine

The purpose of the test execution engine is to produce a verdict for the initial requirement. It takes as inputs the test case produced by the test generator, a test objective, and a mapping describing how to execute SUT interactions used in the test modules.

The architecture of the engine is depicted in Fig. 4. First the test case (a set of XML files) is loaded using the test case loader. Each test process is executed in a separate Java thread. A centralized scheduler implements both the internal communications between the test processes (based on “rendez-vous” and shared variables), and solves the priority conflicts between their actions (according to a predefined policy). Moreover, interactions to be performed on the SUT transit through a Concretisation Wrapper. This component is in charge of transforming these interactions into executable operations on the SUT (depending on the communication medium used, *e.g.* Java RMI). This transformation may also add some parameters omitted at the test module level (for the sake of simplicity), but mandatory for the test execution. Finally, the test selection operation described in Sect. 2.4 is performed by the Objective Engine. When the test execution terminates a verdict is issued and the Logger produces some execution traces that help the diagnostic phase.

4 j-POST at work

We describe in this section the use of j-POST on an example. Tests are designed, generated, executed using the j-POST toolchain to check some properties on a travel agency application [14], called *Travel*. We take as inputs an informal requirement extracted from the functional specification of *Travel* and the application interface. The requirement we choose for the demonstration purpose is informally expressed as “it is impossible to create a mission in *Travel* before being connected”.

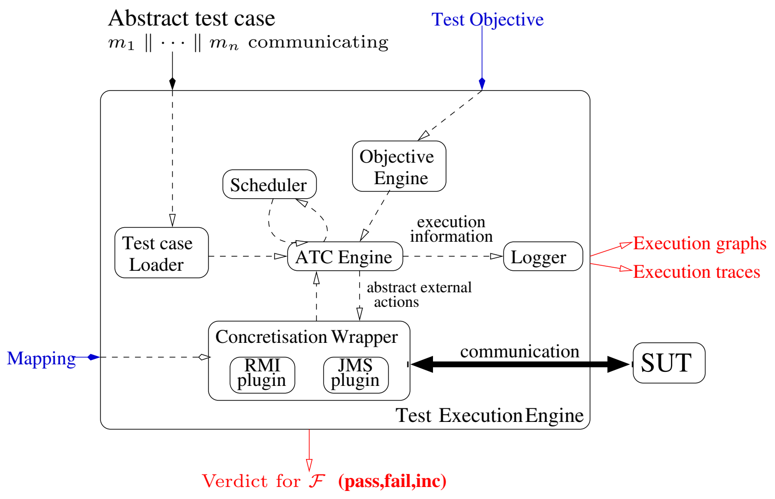

Fig. 4. Abstract view of the j-POST test execution engine

Test design. We start by presenting the test design stage, that is the requirement formalization and the edition of test modules.

Requirement formalization. A possible understanding of our requirement could be that a behaviour in which it is possible to create a mission before performing the identification action is not desired. In other words, we can say that we require no mission creation *until* a connection is open. This informal statement refers to two abstract operations: "create a mission", and "open a connection". In the following we respectively designate these two operations by the predicates $missionCreation()$ and $connection()$. The requirement can be expressed formally by an LTL formula: $(\neg missionCreation()) \; \mathcal{U} \; connection()$.

Test module edition. Test modules have to be created by the user for the predicates $missionCreation()$ and $connection()$. Each of this module should describe:

– how to perform the abstract operation using the *Travel* interface;
– what is the *test verdict* obtained (depending on how *Travel* reacts).

Possible test modules are proposed in Fig. 5, produced with the j-POST test designer. The *connection* test module (left-hand side) contains three possible execution sequences: a correct call to the connection method `identify` (the user is "Falcone", the correct password is "azerty", which corresponds to a registered user of *Travel*), an incorrect one (the password is "qwerty", it is not valid), and an execution where the connection procedure is never called. Note that the call to the `identify()` method returns an identification number which is stored in a *shared* variable (between test components) called `id`. The *createMission* test module (right-hand side) consists of calling the `missionRequest()` method,

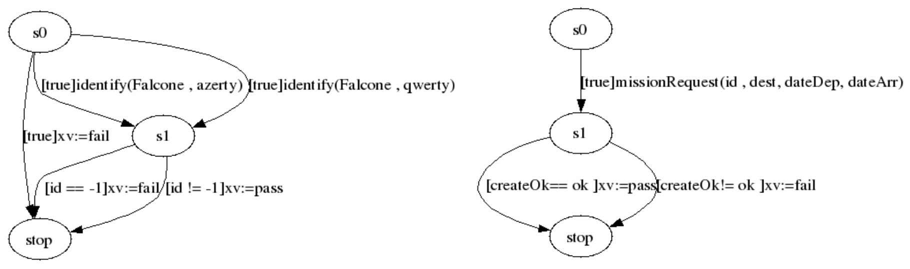

Fig. 5. Test modules for predicates *connection* and *createMission*

supplying the shared variable `id` as an identification number. Depending on the return value (`createOk`), it delivers the corresponding verdict.

Inside the toolchain these modules are represented using an XML format, but, from a practical point of view, they can be written and viewed using the j-POST test designer.

Test generation. The requirement stated, and the test modules designed (Fig. 5), we are now able to perform the test generation. In order to illustrate such a process, we give an insight of the generated test case on Fig. 6. The structure of this test case follows the structure of the formula. It contains a test controller for each operator appearing in the formula (*Until* and *Not*), and a test module for each predicate (*missionCreation()* and *Connection()*). The `testCaseLauncher` is in charge of managing the execution of the testcase and emitting the final verdict. The *c_start* (resp. *c_stop*, *c_loop*, *c_ver*) channels are used by the processes to perform starting (resp. stopping, rebooting, verdict transmission) operations.

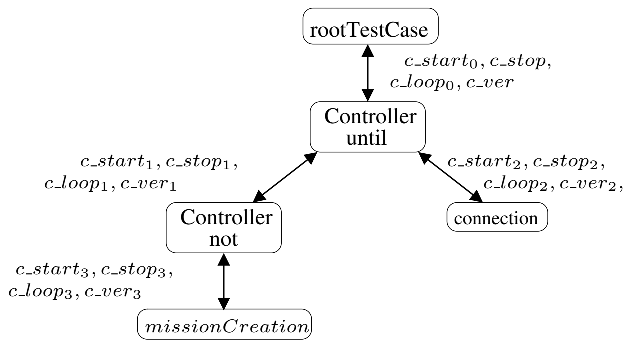

Fig. 6. Test case produced from $\neg(missionCreation)\,\mathcal{U}\,connection$

Test execution. The next operation to perform is to choose a *test objective* in order to restrict the set of potential test executions. Regarding the requirement we consider ("no mission creation until a connection is open"), an interesting objective is to try to *falsify* this requirement in order to exhibit an incorrect behaviour of the software under test. Falsifying such a requirement means for instance producing an execution sequence where :

– the verdict delivered by *missionCreation*() is *pass* (possibly after several previous *fail* results) ;
– in the meantime, the verdict delivered by *connection*() remains always *fail*.

Such a test objective can be obtained from the test modules given on Fig. 5. However, obtaining a *fail* verdict for a connection operation can be fully controlled by the test execution engine (*e.g.*, by supplying an incorrect password), whereas the verdict returned by a mission creation cannot be controlled (it only depends on the SUT behaviour).

Three versions of the *Travel* application have been tested:

– Experiment 1. In the first (erroneous) version of *Travel* a mission creation request is always accepted, therefore our requirement is false (a mission can be created by a non connected user). The test execution engine detects this error (it delivers a *fail* verdict) and produces the test execution traces and graphs for the test case and each module.
– Experiment 2. In the second (erroneous) version of *Travel* a mission creation request is accepted either if the identification number supplied is correct (it corresponds to a return value of a connection request), or if it is the third attempt to create this mission. Therefore our requirement is still false: if a non connected user tries repeatedly to create a mission, it eventually succeeds. This error is detected by the test engine, which delivers a *fail* verdict.
– *Experiment 3.* Finally, the third version of *Travel* always refuses a mission request as long as the identification number supplied is invalid. Thus, the only way for a non connected user to create a mission is to "guess" a correct identification number. This cannot be achieved by our test execution engine, which delivers here a *pass* verdict.

5 Conclusion and perspectives

This paper presents an original approach for property-oriented software testing (POST). Starting from a formula expressed in a trace-based logic, the user first provides a test module (using the test designer) dedicated to each predicate appearing in this formula. The test generation phase then consists of producing a test case as a set of communicating test processes by combining the test modules with some test controllers associated to each logical operator. This test case can be executed by a test engine, able to take into account a test objective to constrain the set of test sequences to execute. This whole testing approach has been implemented in a working tool and applied to some non-trivial case studies. The architecture makes it *open*, and easily allows the toolchain to support new logical formalisms by adding logic plugins.

The main advantage of this approach is that it does not require a "global" behavioural specification of the software under test, as is the case in many model-based testing approaches. In fact the user only needs to make explicit the evaluation of a predicate in the test modules. The test generation phase is therefore rather straightforward and does not suffer from state explosion limitations. Of course, the test case produced may encompass many possible test executions, but the use of test objective allows the user to select the most interesting scenarios. This approach seems particularly relevant to dealing with security or robustness testing, where the functional model of the SUT can be very large (and hence not easily available as a single formal specification), and where the requirements to be checked only concern specific parts of this model. In fact, one of the motivations for this work was the validation of the correct deployment of security policies within the French Politess [15] project.

The *Travel* case study allowed many enhancements for j-POST and opens several research perspectives. In particular, it appears that the design of test modules could be facilitated by the use of abstract domains (*e.g.*, at the test module level one only needs to distinguish between *correct* passwords and *incorrect* ones, without referring to a concrete value). These abstract domains could then be concretized only at the test execution level by selecting relevant values within a concrete domain (which may depend on the SUT's current state). This concretisation could be performed, for instance, according to coverage criteria that could be defined with respect to the requirement under test. It seems particularly worthwile to relate this work with [16].

References

1. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A Test Calculus Framework Applied to Network Security Policies. In: FATES/RV. LNCS 4262 (2006) 55–69
2. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A Compositional Testing Framework Driven by Partial Specifications. In: TestCom/FATES. LNCS 4581 (2007) 107–122
3. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software - Concepts and Tools **17**(3) (1996) 103–120
4. Jard, C., Jéron, T.: TGV: theory, principles and algorithms, A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Software Tools for Technology Transfer (STTT) **6** (2004)
5. Tretmans, J., Brinksma, E.: TorX: Automated Model Based Testing - Côte de Resyste. In: Proceedings of the First European Conference on Model-Driven Software Engineering. (2003) 13–25
6. Belinfante, A., Frantzen, L., Schallhart, C.: Tools for Test Case Generation. In Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A., eds.: Model-Based Testing of Reactive Systems. LNCS 3472 (2004) 391–438

7. Hartman, A.: Model Based Test Generation Tools Survey. Technical report, AGEDIS Consortium (2002)
8. j-POST Reference Page: http://www-verimag.imag.fr/~async/jpost.html. (2007)
9. Schmitt, M., Ebner, M., Grabowski, J.: Test Generation with Autolink and Testcomposer. In: 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'. (2002)
10. AT&T Research: Graph Visualization Software. http://www.graphviz.org (2007)
11. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer-Verlag New York, Inc., New York, NY, USA (1995)
12. Clarke, E., Grumberg, O., Peled, S.: Model Checking. The MIT Press (1997)
13. Kleene, S.C.: Representation of events in nerve nets and finite automata. In Shannon, C.E., McCarthy, J., eds.: Automata Studies. Princeton University Press, Princeton, New Jersey (1956) 3–41
14. Falcone, Y.: A Travel Agency Application. Technical report, Vérimag (2007)
15. Project Politess: ANR-05-RNRT-01301. http://www.rnrt-politess.info (2007)
16. Lestiennes, G., Gaudel, M.C.: Testing processes from formal specifications with inputs, outputs and data types. In: ISSRE, IEEE Computer Society (2002) 3–14