Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

# j-POST: a Java Tool Chain for Property-Oriented Software Testing

*Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, Jean-Luc Richier*

**Verimag Research Report n$^o$ TR-2008-15**

September 23, 2008

# j-POST: a Java Tool Chain for Property-Oriented Software Testing

*Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, Jean-Luc Richier*

September 23, 2008

**Abstract**

j-POST is an integrated tool chain for property-oriented software testing. This tool chain includes, a test designer, a test generator, and a test execution engine. The test generation is based on an original approach which consists in 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. A typical application of j-POST is to check the correct deployment of security policies.

**Keywords:** testing, tool chain, Java, partial specification, requirement, LTL, ERE

**Notes**:

**How to cite this report:**

@techreport { ,
title = { j-POST: a Java Tool Chain for Property-Oriented Software Testing},
authors = { Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, Jean-Luc Richier},
institution = { Verimag Research Report },
number = {TR-2008-15},
year = { 2008},
note = { }
}

## 1 Introduction

j-POST is an integrated tool chain for Property-Oriented Software Testing (POST).

**Property-Oriented Software Testing.** The approach proposed in j-POST relies mainly on an original test generation technique whose theoretical framework is described in [1, 2, 3]. 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 (SUT), 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 a set of *communicating test processes* from this partial specification. Each test process is either an abstract test component or a controller. Then, the test execution engine is able to coordinate the execution of these processes against a distributed Java program, leading to a satisfiability verdict with respect to the given requirement.

**Comparison with classical Model-Based Testing.** This approach offers several advantages over the more classical model-based test generation technique ([4, 5, 6]) implemented in several existing tools (like TGV [7], TorX [8], Autolink [9], see [10] or [11] for a more exhaustive survey). 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 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 tool chain 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 tool chain integrates the whole test process, from the design of the partial specification to the test execution. Note however that the components of this tool chain are loosely coupled, which allows the use of the test execution engine with other test generators (like TGV).

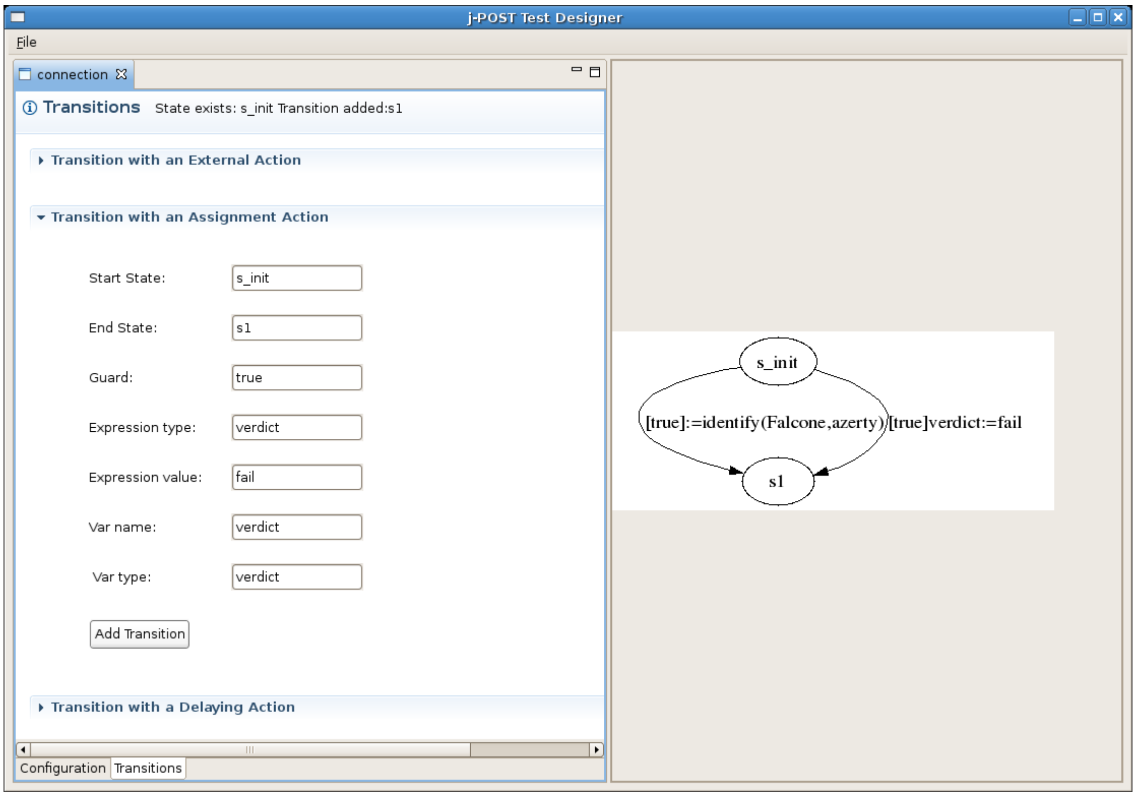
The remainder of this report is organized as follows. The second section presents general information about j-POST. The Sect. 3 illustrates the use of j-POST on a small example. Sect. 4,5,6 are respectively about the test designer, the test generator, and the test execution engine. The Sect. 6 presents some conclusions about j-POST.

**Changes with TR-2007-8** From the last report (TR-2007-8) of j-POST, this version includes the following changes:

- Description of the logical formalisms extensions.
- Description of the newly implemented test objectives.
- Updtating the mapping description.
- Updtating the running example.

## 2 Illustrating the use of j-POST on an example

We describe in this section the use of j-POST on an example. Tests are designed, generated, executed using the j-POST tool chain to check some properties on a travel agency application [12], 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 validate a mission that not have been created first”.



Transitions State exists: s_init Transition added:s1

Transition with an External Action

Transition with an Assignment Action

| | |
|---|---|
| Start State: | s_init |
| End State: | s1 |
| Guard: | true |
| Expression type: | verdict |
| Expression value: | fail |
| Var name: | verdict |
| Var type: | verdict |

Add Transition

Transition with a Delaying Action

Figure 1: Edition of a test module with the designer

## 2.1 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 behavior in which it is possible to validate a given mission before performing the corresponding mission creation operation is forbidden. In other words, we can say that we require no mission validation *until* a corresponding mission creation is performed. This informal statement refers to two abstract operations: “validate a mission”, and “create a connection”. In the following we respectively designate these two operations by the predicates $missionValidation()$ and $missionCreation()$. The predicate $missionCreation()$ takes two “external” (i.e., user supplied) parameters indicating the mission destination and dates. Thus, the requirement can be expressed formally by the following LTL formula:

$$(\neg missionValidation()) \ \mathcal{U} \ missionCreation(NewYork, 01/12/2008, 20/12/2008)$$

**Test module edition.** The test module edition in j-POST is represented on the Fig. 1. The user of the designer adds (left-hand side) transition and the corresponding graph is pictured (right-hand side).

Test modules have to be created by the user for the predicates $missionValidation()$ and $missionCreation()$. 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 described below.

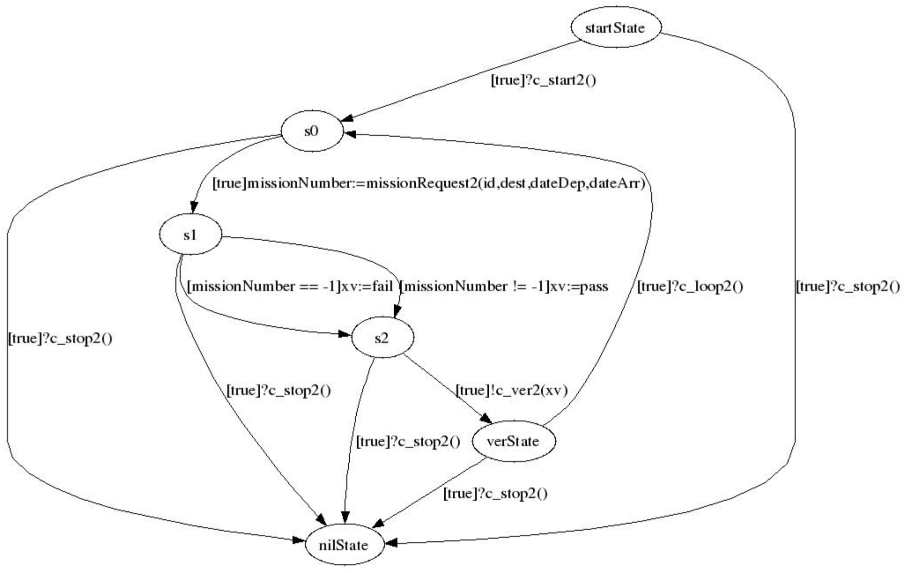


Figure 2: Test module for predicate $missionCreation$

- The $missionCreation$ test module (Fig. 2) essentially performs a call to the `missionRequest2(id, New York,01/12/2008,20/12/2008)` abstract method, where `id` is a parameter indicating the user identity. This value is internal to the *Travel* application, and it is generated when the user opens a connection to *Travel*. This operation, which is mandatory to execute the test case, will be introduced within a *test objective* (see below). Note that `missionRequest2` operation returns a `missionNumber` those value could be either a positive integer, in which case the the returned verdict is `pass`, or a $-1$, in which case the the returned verdict is `fail`. or `pass` otherwise. Variables `id` and `missionNumber` are declared as *shared*, meaning that they can be read and assigned by several test modules. The transitions labelled with `c_start2`, `c_loop2`, `c_ver2` and `c_stop2` are automatically added by the test generator to manage the test execution (they not need to be written by the user).
- The $missionValidation$ test module (Fig. 3) performs a call to the `validMission(id, missionNumber)` abstract method. This call returns either a correct value `ok` in which case the resulting verdict is `pass`, or an incorrect one, in which case it is `fail`. Here again, transitions labelled with `c_start3`, `c_loop3`, `c_ver3` and `c_stop3` are automatically added by the test generator.

Note that inside the tool chain the test modules are represented using an XML format, but, from a practical point of view, the j-POST test designer facilitates their writing and edition.

## 2.2 Test generation

From the formal LTL requirement and the associated test modules we are now able to perform the test generation phase. The (textual) command to use indicates the filename of the formula description (`formula.ltl`) and the destination directory to store the resulting test case (`TC_formulaltl`).

```
java -jar testGeneration.jar -in formula.ltl -out TC_formulaltl
```

In order to illustrate such a generation process, we give an insight of the generated test case on Fig. 4. 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 (*missionValidation()* and *missionCreation()*). The `testCaseLauncher` is in charge of managing the execution of the test case 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.

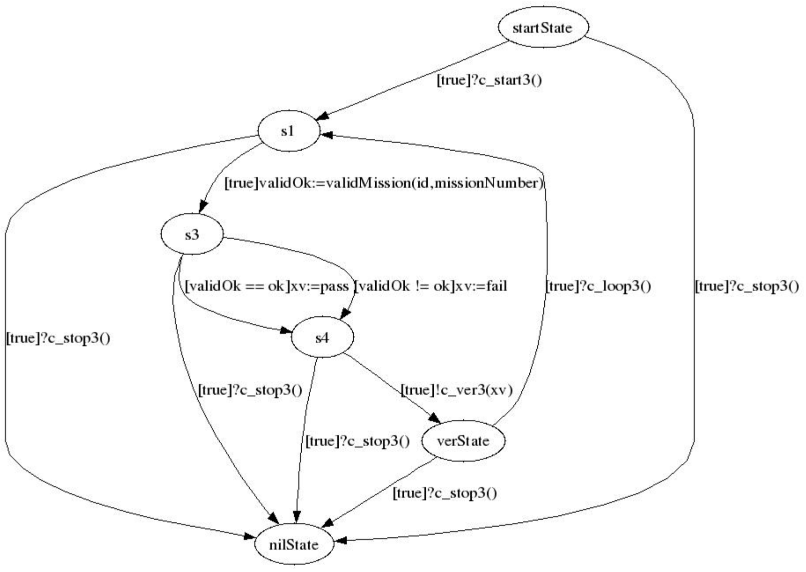


Figure 3: Test modules for predicate $missionValidation$

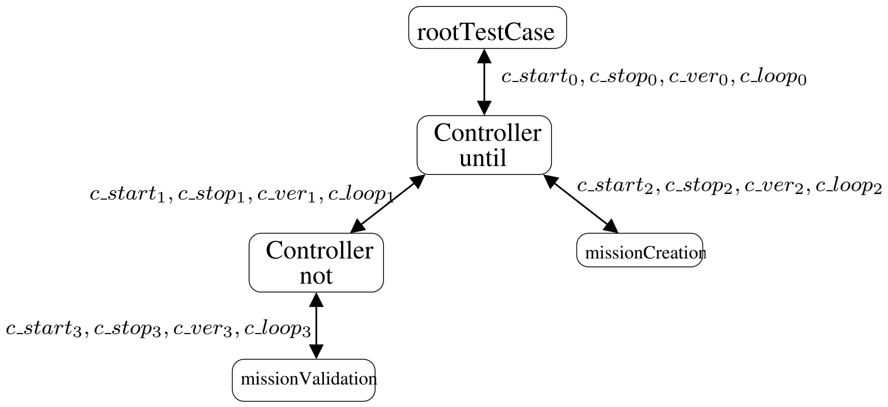

rootTestCase
$c\_start_0, c\_stop_0, c\_ver_0, c\_loop_0$
Controller until
$c\_start_1, c\_stop_1, c\_ver_1, c\_loop_1$
$c\_start_2, c\_stop_2, c\_ver_2, c\_loop_2$
Controller not
missionCreation
$c\_start_3, c\_stop_3, c\_ver_3, c\_loop_3$
missionValidation

Figure 4: Test case produced from $\neg(missionValidation)\,\mathcal{U}\,missionCreation$

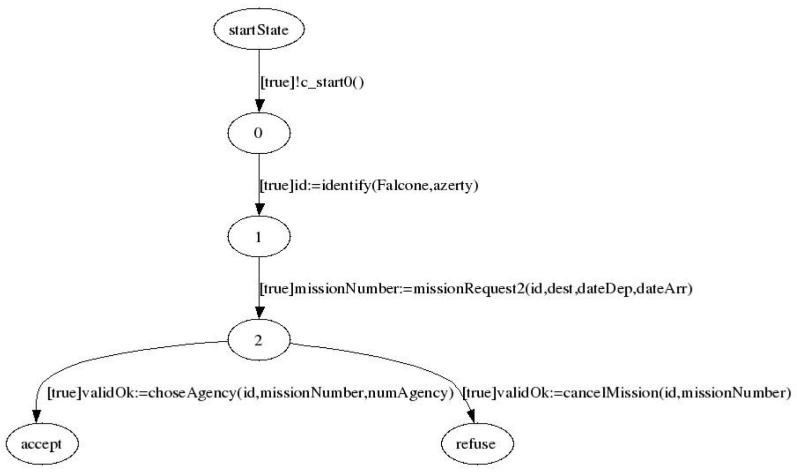

startState
[true]!c_start0()
0
[true]id:=identify(Falcone,azerty)
1
[true]missionNumber:=missionRequest2(id,dest,dateDep,dateArr)
2
[true]validOk:=choseAgency(id,missionNumber,numAgency)
[true]validOk:=cancelMission(id,missionNumber)
accept
refuse

Figure 5: Test objective

## 2.3 Test execution

The next operation to perform is to choose a *test objective* in order to restrict the set of potential test executions. As mentioned before, to create a mission with the *Travel* application needs to obtain first a user identity expressed by an integer value (this how the application interface works). This integer is obtained when a user opens a connection to travel, using a method called `identify`, with a a user name and password as parameters. To enforce the call to this method during test execution, a solution consists in using a *test objective* those purpose is both to guide the test execution through particular scenarios and to "insert" additional interactions with the SUT when they are necessary. This test objective is expressed as a finite state automaton with particular states (*accept* and *reject*) to define expected and unexpected scenarios. We will use here the test objective depicted on figure 5. Roughly speaking, this test objective indicates that an expected scenario is the one (leading to the *accept* state) that successively opens a connection, creates a mission, and chooses a travel agency. If the mission creation fails, then the *accept* state is reached, meaning that the test will be irrelevant (and it will give a non conclusive verdict).

The test objective defined, and the test case generated, one can use the test execution engine to execute the test case against the SUT. The command to use is the following:

```
java -Djava.security.policy=java.policy -jar testEngine.jar -tc TC_formulaltl \
  -mapping mapping.xml  -obj objective.xml -log TC_formulaltl/LOG
```

This command indicates the directory containing the test case (`TC_formulaltl`) the mapping file (`mapping.xml`), the test objective (`objective.xml`) and the destination directory for the log files produced during test execution (`TC_formulaltl/LOG`).

Three experiments have been tested:

- Experiment 1. In the first (erroneous) version of *Travel* a mission validation can be performed even for a non existing (i.e, not previously created) mission. The test execution engine detects this error (it delivers a *fail* verdict) and produces the test execution traces and graphs for the main test process and each test module.

- Experiment 2. In the second experiment we ran the test engine on a correct version of *Travel* but without any test objective. During the test execution no mission creation operation succeeds (`missionRequest2` is called with an incorrect random user id), and the test execution stops with an *inconclusive* verdict due to a timeout expiration (the whole test execution is guarded by a timer).

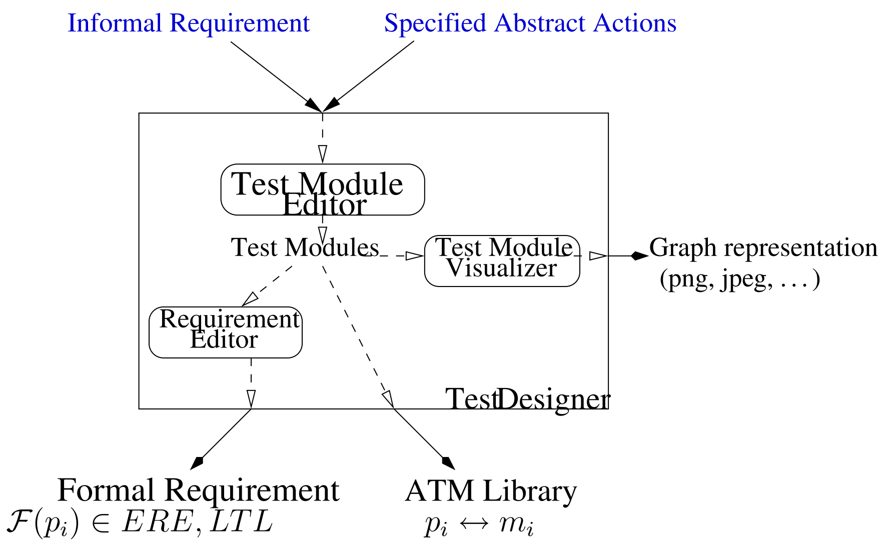


Figure 6: Architecture of the test designer of j-POST

- *Experiment 3.* Finally, executing the test engine on this correct version of *Travel* with the test objective gave (as expected) a *pass* verdict. The log file obtained for the `createMission` test module is given below:

```
/////////////////////////////////////////////
createMission
Path=formulaltl/createMission2Inst
anciation.xml
/////////////////////////////////////////////
* Starting time: Wed Sep 17 17:22:16 MEST 2008
TRANSITION START
startState;[true]?c_start2()
s0;[true]missionNumber:=
      missionRequest2(id,dest,dateDep,dateArr)
s1;[missionNumber != -1]xv:=pass
s2;[true]!c_ver2(xv)
verState;[true]?c_loop2()
s0;[true]missionNumber:=
      missionRequest2(id,dest,dateDep,dateArr)
s1;[missionNumber != -1]xv:=pass
s2;[true]?c_stop2()
nilState;TRANSITION END
createMission> no possible transition, finishing
* Ending time: Wed Sep 17 17:22:17 MEST 2008

The verdict: PASS
```

## 3 Test Design

This section describes how it is possible to design test entities (abstract test case library and requirement) with the j-POST Test Designer (Fig. 6).

### 3.1 Overview

The interface of the SUT contains a set of public variables and methods. Using this input, the user writes a library of abstract test modules, corresponding to high-level operations that can be performed on the SUT. These components can be viewed as terms of a “test calculus” [1]. Roughly speaking, this calculus offers

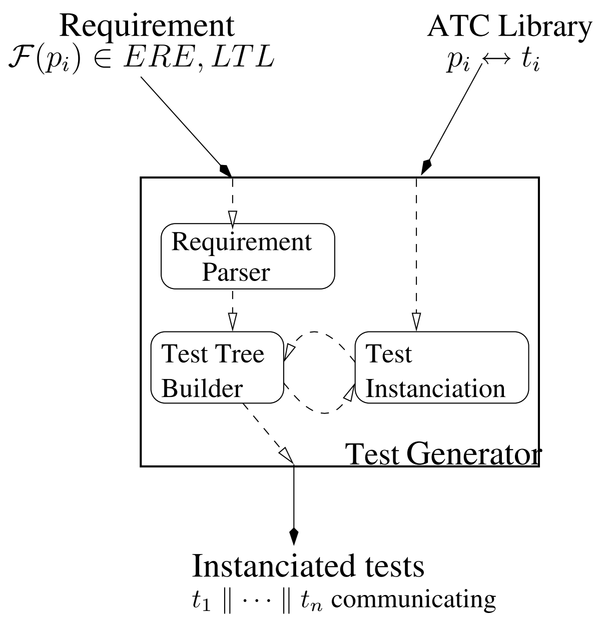


Figure 7: Architecture of the test generator of j-POST

the basic primitives required to compose elementary actions in order to describe more complex behaviours. These primitives include sequential and parallel composition, non-deterministic choice, recursion, and data manipulations. The elementary actions can be either internal actions of the test components (like internal variable assignments) or external calls to the SUT interface. Note that the execution of an external action is considered atomic.

The test designer of j-POST provides a user interface to design these abstract test modules and visualize them in an intelligible way. The XML format has been chosen as an internal representation of these components. We used the Eclipse Modeling Framework [13] to generate a test writer: users of j-POST test designer can produce their test cases by composing interface actions without error-prone manipulation of XML. Also, we provide a test case viewer representing the test case as a labelled transition system. This part of the test designer uses GraphViz [14] to generate the test case image.

## 4 Test Generation

This section describes the functioning of the test generator and the test generation process (Fig. 7).

### 4.1 Overview

The j-POST test generator consists mainly in a test generation function. This function takes as input a formal requirement $\phi$, written in a trace-based logical formalism. It produces a complete test case following a syntax-driven approach:

- For each atomic predicate $P_i$ of $\phi$, an instance of the corresponding abstract test component $t_i$ is produced ;
- For each subformula $\phi = F(\phi_1, \cdots, \phi_n)$ of $\phi$, the test generator function instantiates a (generic) *test controller* dedicated to the operator $F$, to be executed in parallel with the test processes (recursively) obtained for each $\phi_i$. The purpose of the test controller is both to schedule the test execution of the sub-testers and to combine their verdicts in order to produce the verdict associated to $\phi$.

Generic test controllers and test generation algorithms have been defined for different specification formalisms. So far, j-POST supports two common-use formalisms.

- Temporal logics [15] 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. Our case studies have shown however that many concepts of access control security policies fall in the scope of these logics.
- Extended Regular Expressions. Regular expressions [16] are another formalism to define behavior patterns expressed by finite execution traces. They are commonly used and well-understood by engineers. We provide support for for the negation operator to allow the specification of not desired behaviors.

## 4.2 Parsing of the requirement

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 components corresponding to the atomic predicates of the formula, as they are provided by the user. 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 delivers the resulting verdict.

The analysis of the formula (expressing the requirement) is performed thanks to a syntactic analyzer. To realize the parsing, we chose Java-CC [17]. This a Java parser/scanner generator. It generates a set of Java classes by analyzing a set of grammar rules (see Sect. B) describing the language to be analyzed and the Java code inserts to be performed.

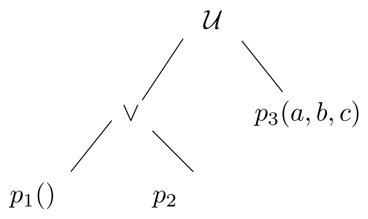


Figure 8: Abstract syntax tree corresponding to $(p_1() \vee p_2())\mathcal{U}p_3(a,b,c)$

## 4.3 Implementation of the test generation function (Test tree building + Test instantiation)

To apply the test generation principle ([3], Sect. 3), we build the abstract syntax tree. For instance, if we consider the LTL expression: $(p_1() \vee p_2())\ \mathcal{U}\ p_3(a,b,c)$, the abstract tree is represented on Fig. 8. In the following, we illustrate only the implementation for the LTL version of $GenTest$, the principle is identical for the EREs. Thanks to the syntactic analyzer, we built the syntactic tree. In order to represent the tree in memory, we defined the following classes:

- Node.java: This is an abstract class containing all the information shared by the tree nodes. Each of the following classes representing the nodes inherit from this class. It rules the implementation of the needed methods for the generation.
- FormulaNode.java: Represents the LTL *always* and *eventually* unary controllers.
- PredicateNode.java: Represents the test predicate of the formula.
- BinaryOperationNode.java: Represents the binary *and*, *or*, *imply* and *until* controllers.

Each of these classes (representing the controllers or the predicates) owns one or several attributes representing their operands. As the $GT$ function is recursive, it is defined in each of the controller class, the $Test$ function in the class PredicateNode and $GenTest$ in the class Node.

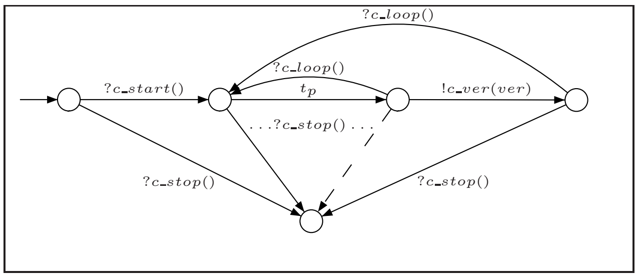


Figure 9: Instantiation of an abstract test component $t_p$

As seen before, the test processes (predicates, and controllers) are represented in XML. Each of the test being instantiated by $GenTest$, their structure or parameters are modified. We used JDOM [18], a tool permitting the manipulation and modification of XML data.

For each of the requirement formalism, the principle is:

- $GenTest$: Creates the master controller managing the test process at the highest level in the tree. In a second time, it calls a generation method associated to this process (with a fresh communication channel name to instantiate it).
- $GT(F^n(\phi_1, \cdots, \phi_n), cs)$: Takes as parameter a communication channel $cs$ and generates one new communication channel per operand of $F^n$. The generic test controller associated to operator $F^n$ is then instantiated with these channel names, and the generation method is (recursively) called for each operand of $F^n$ (with the corresponding channel name as parameter).
- $Test(t_p, \{c\_start, c\_stop, c\_loop, c\_ver\})$ Takes a channel number and generates control states and transitions of this predicate and implements it in the structure (see Fig. 9).

Once this stage is finished, we obtain a list of XML files representing the instantiated test processes. They can be used then as an entry for the test execution engine. Furthermore, the test execution needs the existence of two supplementary files. The first one is the list of all actions that each of the tests process can perform. The second one is the list of the channel used in the potential internal communication of the test processes. These files are also defined in a XML scheme. These two files are produced during the tree browsing by $GenTest$.

## 5 Test Execution

This section describes how tests are executed with the j-POST test execution engine (Fig. 10).

### 5.1 Overview

The purpose of the test execution engine is to produce a verdict for the initial requirement. To do so, it uses the complete test case produced by the test generator and a *test objective*.

**Test selection using dynamic test objectives.** The test case produced by the test generator may 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 $\varphi$: it does not help to select the *interesting* test executions that are likely to exhibit an incorrect behavior of the SUT. To solve this problem we propose to use *behavioral test objectives*, already introduced in several model-based testing tools (*e.g.* in [7, 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

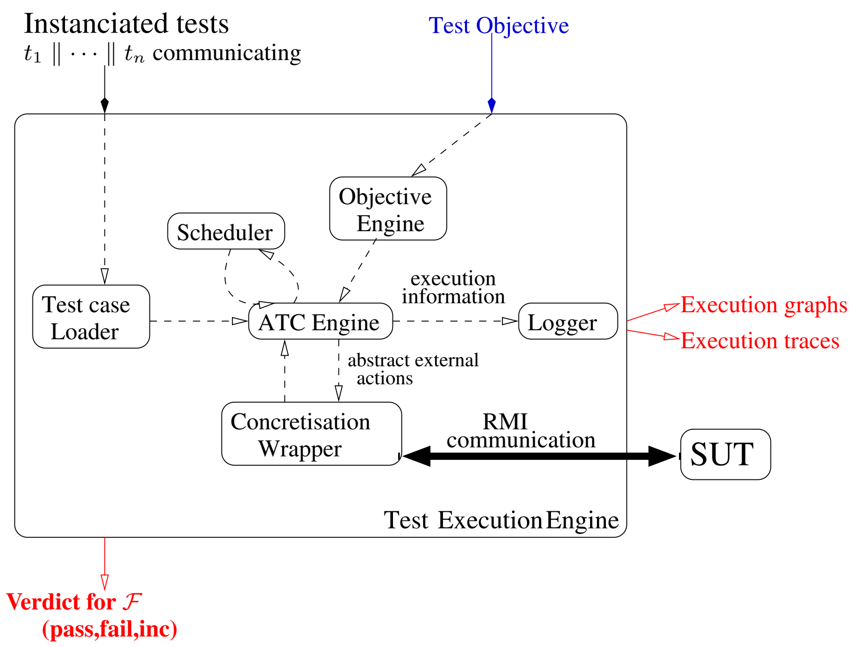


Figure 10: Architecture of the test execution engine of j-POST

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. A formal definition of this notion of test objectives within the j-POST framework can be found in [19].

**Multi-threaded test execution with "on-the-fly" concretisation.** Once the test process has to perform an external action, this action is concretised "on-the-fly". To do so, the user furnishes a mapping between the external action and the SUT interface actions. Using this mapping, the test execution engine concretises the action and produces the corresponding interface call. Each test process produced by the test generator is executed in a separate Java thread. It also allows a user to define scheduling policies and priority order between each type of interaction.

## 5.2 Functioning principle

The purpose of this tool is to execute a set of test processes to decide a verdict allowing to validate or not the requirement used to generate the tests. A set of communicating test processes is used as an input of the tool. In the first time, the files containing the channel, and the action list are analyzed. Then in a second time, the file of each of the given test process is analyzed in order to the corresponding automaton in memory. Each of the test process is modeled by a Java thread in memory. As so, to launch the corresponding test process, we just have to launch the thread.

### 5.2.1 Construction and evaluation of the test automata

Each of the test processes owns a memory containing the set of variables and parameters. There is also a set of states and transition which actions are of several types (see the next section). Moreover, the test processes detain a shared memory allowing to stock common values. When a test process is launched, it starts evaluate the guard of the transitions which departure state is the initial state. The guard are boolean expression, their evaluation is ensured by a parser generated by Java-CC. The grammar of the boolean expression can be found in Sect. A. The test process is going to obtain the list of the possible transitions and will be able to choose later the one it will perform. The next subsection explains how this choice is done. Once the action determined, the test process tries to realize it. If it succeeds, the stage change is executed and the execution is pursued following the same principle. If not, the test process makes another choice with the remaining transitions and tries to execute the recently chose action. If no action is performed well, the test process stops its execution.

### 5.2.2 Test selection with respect to test objective

This section describes how the test engine proceeds to select a particular test sequence among the ones offered by the current test case. More precisely it explains how the use of a test objective helps to both take into account some scheduling directives (internal to the test case execution), and to enrich the test execution with extra interactions between the tester and the SUT.

First of all, let us recall the three different kinds of actions a test process can perform:

**Internal operations.** These actions are local to the test process, typically data manipulations, or temporisation actions.

**Internal communications.** These actions are synchronization operations with other processes. The process needs to receive or emit a signal with parameters or not through a specified channel. All processes that want to synchronize using this channel wait that other participants are in the rendez-vous. Depending on the policy, the processes will have to wait for all expected participants or only for a subset. At least, one emission and one reception is required in order to realize the communication.

**External interactions.** These actions are used to communicate with the SUT (e.g., method call). A concretization step is required to translate first the abstract interaction (as it appears in the test process) by a concrete one (as it is expected by the SUT). This step is described in the next subsection.

Several policies can be considered to favor one kind of actions or an other. In order to stay as modular as possible, a specific class encodes the policy that one might want to choose. In the default policy, the process privileges internal actions among all the other kinds, then external actions are preferred, and finally the internal communications. Conflicts occurring at a given level (i.e, two test processes compete to perform both an internal action) are solved by the underlying default Java scheduler (since each test process is implemented by a distinct Java thread). In practice a test process sends its set of pending actions to the j-POST scheduler, and waits for its notification to proceed. It is therefore blocked as long as the scheduler decides to notify it.

This default scheduling policy can be modified by means of a test objective. Formally, a test objective is a deterministic automaton with special states noted "Accept" and "Reject". Its transition are labelled either with internal communications or with external interactions. Moreover, a priority relations between outgoing labels can be associated to each states. Intuitively, execution sequences leading to "Accept" states denote expected test scenarios whereas execution sequences leading to "Reject" states denote unwanted ones. Note that these execution sequences may contain "extra" communication interactions, i.e., interactions with the SUT that do not appear in the test case itself (provided they are in the test interface of the SUT).

The test objectives are used by the scheduler for test selection according to the following rules:

- if some actions proposed by the test processes match the labels of the outgoing transitions of the current state of the test objective, then one of the most prioritary ones is (randomly) chosen, and the corresponding test process is notified ;
- if the intersection between the actions proposed by the test processes and the labels of the outgoing transitions of the current state of the test objective is empty then two cases can occur:
  - either there exists some (extra) communication interactions offered by the test objective in the current state, and therefore one of the most prioritary ones is (randomly) executed ;
  - otherwise, the conflict is solved by the default Java scheduler.

```
<interaction guard="true" type="emission_reception" scope="external">
        <call name="identify">
          <param type="string" value="A_Possible_Login" />
          <param type="string" value="A_Password" />
        </call>
        <var name="id" type="integer" />
</interaction>
```

Figure 11: Example of how an external action is XML-coded

## 5.3 Concretisation of external interactions

The external interactions are represented in an abstract form in the test processes. This abstraction allows to be independent towards the tested architecture. A concretisation phase is therefore needed for these actions to become executable on the SUT. To be as general as possible the concretisation stage is independent from the test execution engine. It uses a mapping allowing the translation of external interactions. For now, the concretisation towards Java RMI [20] is supported. We are studying how other technologies can be used with our test execution engine. One seems interesting, JMS [21] is a technology offering asynchronous message passing.

### 5.3.1 Modelization of external actions

The possible kinds of action for an external action are:

- emission: corresponds to a call on the SUT. No answer is waited
- reception
- emission-reception: the test processes performs a call on the SUT and a return value is waited.

On Fig 5.3.1 is an example of XML code describing an external action. In order for an action to be external, the scope attribute value must be **external**. The type is one of those predefined. In the provided example, the test process will process the abstract action "*identify*", to do so, it gives two *string* parameters and waits for a return value of type *integer* which is supposed to be stored in the variable *id*.

In order for an action external action to be executed, the scope attribute value has to be *external*. The type is one of those previously defined. In this example, the test process will perform the abstract action "*identify*", to do so, it gives two parameters of type *string* and waits for a return value of type *integer* that will be stored in the *id* variable.

### 5.3.2 Elements needed for the mapping

The mapping principle is to translate an abstract action in a concrete one in an easy and elegant way. It is needed to know the targeted architecture. For now, j-POST supports black box testing via RMI technology. The elements for which a concretisation is necessary are:

- Call name: The possible calls are the available methods in the remote interfaces made available by the SUT. It is so needed to define the interfaces, their locations, and the callable methods.
- Types of the input parameters and returning values. A correspondence is needed between abstract types and their equivalent in the target value. For now the abstract corresponds simply to the equivalent in Java. The XML scheme is given in Sect. A.

In this file, an interface, named diag is declared. The url corresponds to the object location. In the RMI technology, to get a distant object, it is needed to use *rmiregistry*: a directory listing all available remote interfaces. From the interface name and the url, the rmiregistry will return a reference on the remote object.

```
<mapping>
   <interface name="diag" url="rmi://localhost/" packageName="mappFile" className="Diag">
        <method abstractName="choseAgency" realName="choseAgency" arity="3" >
        <msgReturn concreteValue="Wrong number, try again." abstractValue="pasOk" />
        <msgReturn concreteValue="Incorrect status of travel." abstractValue="pasOk" />
        <msgReturn concreteValue="Wrong number." abstractValue="pasOk" />
        <msgReturn concreteValue="Agency correctly chosen" abstractValue="ok" />
        </method>
   </interface>
</mapping>
```

Elements of type *method* are declared for the *diag* interface. These elements correspond to the callable methods. In these elements one could indicate the abstract name (the name used inside the test process), the concrete name (the one appearing within the SUT interface) and the method arity. Furthermore, parameter values can also be mapped in order to associate abstract values (i.e., used in the test process) with concrete ones (i.e., actually returned by the SUT).

## 6 Conclusion

This report presents a Java tool chain for testing properties on software, funded on an original approach. The test generation is driven by a formal requirement and works from partial specifications and algebraic test composition. The development architecture allows extensions by adding logical formalisms via logic plugins. One of the motivation of this approach is to validate the correct deployment of security policies, *e.g.* in the Politess [22] project.

# A XML scheme descriptions

## A.1 Test process description

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <!-- Tag testCase -->
 <xs:element name="testCase">
  <xs:complexType>
   <xs:sequence>
    <!-- param marker : parameters of the testCase -->
    <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="channelList" minOccurs="0" maxOccurs="3">
        <xs:complexType>
         <xs:sequence>
         </xs:sequence>
         <xs:attribute name="startC" type="xs:string" use="required" />
         <xs:attribute name="stopC" type="xs:string" use="required" />
         <xs:attribute name="loopC" type="xs:string" use="required" />
         <xs:attribute name="verC" type="xs:string" use="required" />
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required" />
      <xs:attribute name="type" use="required" >
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:enumeration value="channelSet" />
         <xs:enumeration value="verdict" />
         <xs:enumeration value="boolean" />
         <xs:enumeration value="integer" />
         <xs:enumeration value="string" />
        </xs:restriction>
       </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="value" type="xs:string" use="optional" />
     </xs:complexType>
    </xs:element>

    <!-- declare marker : declaration of a variable -->
    <xs:element name="declare" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required" />
      <xs:attribute name="type" use="required" >
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:enumeration value="verdict" />
         <xs:enumeration value="boolean" />
         <xs:enumeration value="integer" />
         <xs:enumeration value="string" />
        </xs:restriction>
       </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="value" type="xs:string" use="optional" />
      <xs:attribute name="shared" type="xs:boolean" use="optional" />
     </xs:complexType>
    </xs:element>
```

```
<!-- Tag state : declaration d'un etat de l'automate -->
<xs:element name="state" maxOccurs="unbounded">

 <xs:complexType>
  <xs:sequence>
   <!-- Tag transition : description d'une transition de l'automate -->
   <xs:element name="transition" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <!-- Tag interaction : description d'une action de la transition -->
      <xs:element name="interaction">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="channel" type="xs:string" minOccurs="0" />
         <xs:element name="expression" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
           <xs:sequence>
           </xs:sequence>
           <xs:attribute name="type" use="optional" >
            <xs:simpleType>
             <xs:restriction base="xs:string">
              <xs:enumeration value="verdict" />
              <xs:enumeration value="boolean" />
              <xs:enumeration value="integer" />
              <xs:enumeration value="string" />
             </xs:restriction>
            </xs:simpleType>
           </xs:attribute>
           <xs:attribute name="value" type="xs:string" use="required" />
          </xs:complexType>
         </xs:element>

         <!-- For external action -->
         <xs:element name="call" minOccurs="0">
          <xs:complexType>
           <xs:sequence>
             <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
               <xs:sequence>
               </xs:sequence>
               <xs:attribute name="type" use="required" >
                <xs:simpleType>
                 <xs:restriction base="xs:string">
                  <xs:enumeration value="channelSet" />
                  <xs:enumeration value="verdict" />
                  <xs:enumeration value="boolean" />
                  <xs:enumeration value="integer" />
                  <xs:enumeration value="string" />
                 </xs:restriction>
                </xs:simpleType>
               </xs:attribute>
               <xs:attribute name="value" type="xs:string" use="optional" />
              </xs:complexType>
             </xs:element>
           </xs:sequence>
           <xs:attribute name="name" type="xs:string" use="required" />
          </xs:complexType>
         </xs:element>
```

```
             <xs:element name="var" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
               <xs:sequence>
               </xs:sequence>
               <xs:attribute name="type" use="required" >
                <xs:simpleType>
                 <xs:restriction base="xs:string">
                  <xs:enumeration value="verdict" />
                  <xs:enumeration value="boolean" />
                  <xs:enumeration value="integer" />
                  <xs:enumeration value="string" />
                 </xs:restriction>
                </xs:simpleType>
               </xs:attribute>
               <xs:attribute name="name" type="xs:string" use="required" />
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <xs:attribute name="type" use="required" >
             <xs:simpleType>
              <xs:restriction base="xs:string">
               <xs:enumeration value="emission"/>
               <xs:enumeration value="reception" />
               <xs:enumeration value="emission_reception" />
               <xs:enumeration value="affectation" />
              </xs:restriction>
             </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="scope" use="required" >
             <xs:simpleType>
              <xs:restriction base="xs:string">
               <xs:enumeration value="internal"/>
               <xs:enumeration value="external" />
              </xs:restriction>
             </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="guard" type="xs:string" use="required" />
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="nextState" type="xs:string" use="required" />
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required" />
      <xs:attribute name="initial" type="xs:boolean" use="optional" />
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="id" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
</xs:schema>
```

## A.2 Channel description

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="channelList">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="channel" maxOccurs="unbounded" >
     <xs:complexType>
      <xs:sequence>
       <xs:element name="type" minOccurs="0" maxOccurs="unbounded" >
        <xs:complexType>
         <xs:sequence>
         </xs:sequence>
         <xs:attribute name="name" use="required" >
          <xs:simpleType>
           <xs:restriction base="xs:string">
            <xs:enumeration value="verdict" />
            <xs:enumeration value="boolean" />
            <xs:enumeration value="integer" />
            <xs:enumeration value="string" />
           </xs:restriction>
          </xs:simpleType>
         </xs:attribute>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required" />
      <xs:attribute name="arity" type="xs:integer" use="required" />
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

## A.3 Action description

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="actionList">
  <xs:complexType>
    <xs:sequence>
     <xs:element name="interaction" maxOccurs="unbounded">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="channel" type="xs:string" minOccurs="0" />
        <xs:element name="expression" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
          <xs:sequence>
          </xs:sequence>
          <xs:attribute name="type" use="optional" >
           <xs:simpleType>
            <xs:restriction base="xs:string">
             <xs:enumeration value="verdict" />
             <xs:enumeration value="boolean" />
             <xs:enumeration value="integer" />
             <xs:enumeration value="string" />
            </xs:restriction>
           </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="value" type="xs:string" use="required" />
         </xs:complexType>
        </xs:element>
        <!-- For external action -->
        <xs:element name="call" minOccurs="0">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
             <xs:sequence>
             </xs:sequence>
             <xs:attribute name="type" use="required" >
              <xs:simpleType>
               <xs:restriction base="xs:string">
                <xs:enumeration value="channelSet" />
                <xs:enumeration value="verdict" />
                <xs:enumeration value="boolean" />
                <xs:enumeration value="integer" />
                <xs:enumeration value="string" />
               </xs:restriction>
              </xs:simpleType>
             </xs:attribute>
             <xs:attribute name="value" type="xs:string" use="optional" />
            </xs:complexType>
           </xs:element>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="required" />
         </xs:complexType>
        </xs:element>
        <xs:element name="var" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
          <xs:sequence>
          </xs:sequence>
          <xs:attribute name="type" use="required" >
           <xs:simpleType>
            <xs:restriction base="xs:string">
             <xs:enumeration value="verdict" />
             <xs:enumeration value="boolean" />
             <xs:enumeration value="integer" />
             <xs:enumeration value="string" />
            </xs:restriction>
           </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="name" type="xs:string" use="required" />
         </xs:complexType>
        </xs:element>
       </xs:sequence>
       <xs:attribute name="type" use="required" >
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="emission"/>
          <xs:enumeration value="reception" />
          <xs:enumeration value="emission_reception" />
          <xs:enumeration value="affectation" />
         </xs:restriction>
        </xs:simpleType>
       </xs:attribute>
       <xs:attribute name="scope" use="required" >
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="internal"/>
          <xs:enumeration value="external" />
         </xs:restriction>
        </xs:simpleType>
       </xs:attribute>
       <xs:attribute name="guard" type="xs:string" use="required" />
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
 </xs:element>
</xs:schema>
```

## B Concrete syntax of the requirement formalisms

### B.1 Linear Temporal Logic

The abstract syntax of LTL (Linear Temporal Logic) is given by the following grammar, where $p_i(\overline{x})$ designates an atom expressed by a parameterized action predicate:

$$\varphi \ ::= \ \neg\,\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi \wedge \varphi \mid \Box\varphi \mid \Diamond\varphi \mid p_i(\overline{x})$$

The concrete syntax we considered in JavaCC was the following:

```
Spec ::= Formula
Formula ::= Mod Term | Term
Term ::= Term OpBin Factor | not Factor | Factor
Factor ::= Predicate | (Formula)
OpBin ::= and | or | => | until
Mod ::= always | eventually
Predicate ::= name (ListParam)
ListParam ::= name EndListParam | epsilon
EndListParam ::= , name | epsilon
```

### B.2 Extended Regular Expressions

The syntax of ERE (Extended Regular Expressions) is given by the following grammar, where $p_i(\overline{x})$ designates an atom expressed by a parameterized action predicate:

$$\varphi \ ::= \ \texttt{not}\ \varphi \mid \varphi + \varphi \mid \varphi\ \cdot\ \varphi \mid \varphi^* \mid p_i(\overline{x})$$

The concrete syntax we considered in JavaCC was the following:

```
Spec ::= Term
Term ::= Term OpBin Factor | not Factor | Factor * | Factor
Factor ::= Predicate | (Term)
OpBin ::= + | .
Predicate ::= name (ListParam)
ListParam ::= name EndListParam | epsilon
EndListParam ::= , name | epsilon
```

## C About options of j-POST binaries

This section descibres the options

### C.1 Test generator

The option list can be obtained by launching the executable JAR file and providing the `--help` option.

```
ruitor:~/workspace/jpost2/dist/testGenerator> java -jar testGenerator.jar --help
Usage : java -jar testGenerator.jar options
         -in,-input The input path of the requirement
         -out,-output The output path
                      if omited, the output path is determined automatically
         -log [path] log in path (the path to the directory)
         -v use verbose mode
         --help display this help message
         where testGenerator.jar is the name of this JAR file
```

The options that can be provided to the Test Generator are rather simple:

- Option `-in` or `-input` is used to indicate the path to the requirement file.
- Option `-out` or `-output` is used to indicate the path to the output directory for the test case.
- Option `-log` is used to indicate to log information of the process of generation to a log file indicated in argument.
- Option `-v` indicates to the tester to use verbose mode. In this case more information about the generation is displayed on the console.
- Option `--help` is used to display the available options.

## C.2 Test engine

The option list can be obtained by launching the executable JAR file and providing the `--help` option.

```
ruitor:~/workspace/jpost2/dist/testEngine> java -jar testEngine.jar --help
Usage : java -Djava.security.policy=java.policy -jar Test-Engine.jar options
         -tc [path] : indicates the path of the test case
         -map|-mapping [path] : indicates the path of the mapping file
         -obj [path] -> Use the test objective to drive the execution
         -log [path] : the path to the directory used to log
         -graph : produce a graph illustrating the execution
         -debug|-d [path] : the path to the directory use to debug
         -verbose|-v use verbose mode
         --help : displays this help message
```

## References

[1] Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A Test Calculus Framework Applied to Network Security Policies. In: FATES/RV. (2006) 1, 3.1

[2] Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A Compositional Testing Framework Driven by Partial Specifications. In: TESTCOM/FATES. (2007) 1

[3] Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A partial specification driven compositional testing method. Technical Report TR-2007-04, Vérimag Research Report (2007) 1, 4.3

[4] Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools **17** (1996) 103–120 1

[5] ISO/IEC 9946-1: OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework. International Standard ISO/IEC 9646-1/2/3 (1992) 1

[6] Brinksma, E., Alderden, R., Langerak, R. Van de Lagemaat, J., Tretmans, J.: A Formal Approach to Conformance Testing. In De Meer, J., Mackert, L., Effelsberg, W., eds.: Second International Workshop on Protocol Test Systems, North Holland (1990) 349–363 1

[7] 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) 1, 5.1

[8] 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 1

[9] Koch, B., Grabowski, J., Hogrefe, D., Schmitt, M.: Autolink: A tool for automatic test generation from sdl specifications. wift **00** (1998) 114 1, 5.1

[10] Hartman, A.: Model Based Test Generation Tools Survey. Technical report, AGEDIS Consortium (2002) 1

[11] 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. Volume 3472 of Lecture Notes in Computer Science., Springer (2004) 391–438 1

[12] Falcone, Y.: A Travel Agency Application. Technical report, Vérimag (2007) 2

[13] The Eclipse Foundation: Eclipse Modelling Framework. http://www.eclipse.org/modeling/emf (2007) 3.1

[14] AT&T Research: Graph Visualization Software. http://www.graphviz.org (2007) 3.1

[15] Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer-Verlag New York, Inc., New York, NY, USA (1995) 4.1

[16] 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 4.1

[17] SUN Java.net: Java-CC. http://javacc.dev.java.net (2007) 4.2

[18] Jason Hunter: JDOM. http://www.jdom.org (2007) 4.3

[19] Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: j-POST: a Java Toolchain for Property-Oriented Software Testing. In: Model-Based Testing (MBT). (2008) 5.1

[20] Java Remote Method Invocation: Java RMI. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp (2007) 5.3

[21] Java Message Service: JMS. http://java.sun.com/products/jms/ (2007) 5.3

[22] Project Politess: ANR-05-RNRT-01301. http://www.rnrt-politess.info (2007) 6