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-2007-7

April 25, 2008

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

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

April 25, 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-2007-7},
year = { 2007},
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.

2 General information about j-POST

This section provides a quick overview on j-POST: we describe the whole principle of j-POST, the history of the tool chain, and a quick start usage guide.

2.1 Principle

An overview of the functioning principle of j-POST is depicted on Fig. 1. The user of j-POST wants to test an application for which the interface is available. Providing it to the test designer (step 1), the user is able to elaborate first an abstract test component library by combining the actions offered by the SUT interface. Secondly, the user establishes one (or more) requirement(s). To do so, he uses as elementary predicate each property for which he designed a test (available in his library). These two freshly made test entities are run by the test generator (step 2). The result is the production of a set of instantiated communicating test components which are purposed to test the requirement. To be launched by the test engine (step 3), the user can provide a test objective. As so, the test engine uses it to guide the test execution and select the desired execution among the potential multiple execution described in the generated tests.

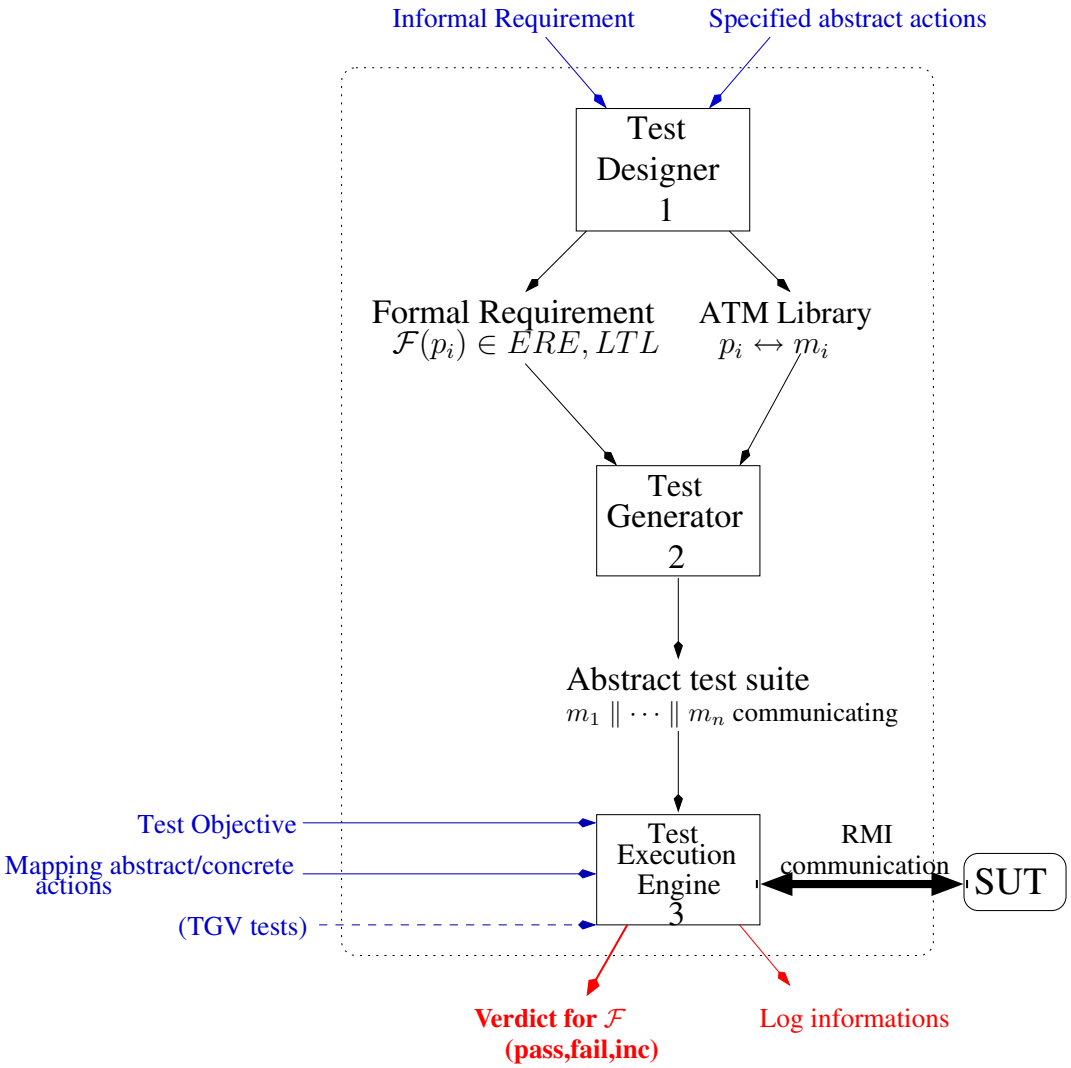

Figure 1: Abstract view of the j-POST testing tool chain

2.2 History and changelog

Here, we list the the evolution history of the different versions of j-POST for each component of the tool chain.

2.2.1 Test designer

For the test designer component the history is the following:

- 02 November: version 0.2 released
  - Full inclusion of the test case editor as an Eclipse EMF plugin.
  - Corrected a bug with the output path of the test case image.
  - Improved the distribution package.
- 01 October: version 0.1 released: now publicly available.
- 15 June 07: first beta version.

2.2.2 Test generator

For the test generator component the history is the following:

- 19 November: version 0.3 released
  - Support for Extended Regular Expression as requirement formalism
- 02 November: version 0.2 released
  - Improved the distribution package
- 01 October: version 0.1 released: now publicly available.
- 15 June 07: first beta version.

2.2.3 Test execution engine

For the test execution engine component the history is the following:

- 09 November: version 0.2.1 released.
  - Improved graphs of execution logs
- 02 November: version 0.2 released.
  - Corrected a bug in the communication between test processes.
  - Improved the distribution package.
- 01 October: version 0.1 released: now publicly available.
- 15 June 07: first beta version.

2.3 Installation and usage

The three tools of j-POST are freely available separately in three archives: TestDesigner.tar.gz, TestGenerator.tar.gz, TestEngine.tar.gz. The tool chain is available on the j-POST web site [12].

2.3.1 Test designer

With the graphic version Just run the executable file contained in the specific version for your operating system. Then edit the preference, in the preference menu.

With the command line version

Installation. In order to use the test designer, you must have available on your system the following applications.

- Eclipse EMF [13] is needed by the TestDesigner-Editor.
- DOT is needed for the TestDesigner-Visualizer. Dot is part of the GraphViz [14] distribution.

In order to install the test designer component, please follow the installation instructions:

1: Decompress the package **TestDesigner.tar.gz**.

Installation of the TestDesigner-Editor

2a: Put the plugins directory in your Eclipse-EMF directory

3a: Re-launch Eclipse. The plugin should appear in About Eclipse SDK→Plugins Details list

Installation of the TestDesigner-Visualizer

2b: Set the path to the dot executable on your system. Edit the 'preferences' file and indicate the path to the dot executable.

Generally:

LINUX: /usr/bin/dot

MAC OS X: /sw/bin/dot

Note that this step is optional as one can specify the DOT path in command line (use the -help option for more details).

3b: Create an input directory for your XML test cases.

4b: Put the test cases in this directory.

Usage, running instructions

TestDesigner-Visualizer. There are two ways for launching the j-POST Test Designer-Visualizer.

- Launch the (UNIX) script designTest, to do so:

  launchTest INPUTDIR OUTPUTDIR

  where:

  * INPUTDIR is the name of the directory containing the test case to be displayed.
  * OUTPUTDIR is the name of the directory containing the result of the design.

- Use directly the jar TestDesigner.jar providing your desired options. Type

  java -jar TestDesigner.jar –help

  for the description of the available options.

TestDesigner-Editor. To use the editor, follow these instructions:

1 Create a new project in your workspace

2 Create a new TestCase Model:

* Right-Click on your project
* Select New→Other...
* Example EMF Model with creation wizard→TestCase Model
* Set a name for your test case

3 Edit your freshly made test case, by using the *New Child* and *New Sibling* commands

2.3.2 Test generator

With the graphic version Just run the executable file contained in the specific version for your operating system. Then edit the preference, in the preference menu.

With the command line version

Installation. Please follow these steps to install the test generator.

1 Decompress the package **TestGenerator.tar.gz**.

2 Put the test cases in the ressource directory.

3 Indicate a requirement in a file.

Usage, running instructions. There are two ways for launching the j-POST Test Generator.

- Launch the (UNIX) script genTest which contain default option, to do so:

  genTest REQUIREMENT

  where REQUIREMENT is the path to the requirement file for the test generation.
  The default option are:

  * output path: output
  * no logging

- Use directly the jar TestGenerator.jar providing your desired options. Type

  java -jar TestGenerator.jar -help

  for the description of the available options.

2.3.3 Test engine

With the graphic version Just run the executable file contained in the specific version for your operating system. Then edit the preference, in the preference menu.

With the command line version

Installation.

1 simply decompress the package **TestEngine.tar.gz**.

2 copy your test case directory at the location of your choice (for example in the same directory).

Usage, running instructions. There are two ways for launching the j-POST Test Engine.

- Launch the (UNIX) script launchTest, to do so type:

  launchTest TESTCASE

  where TESTCASE is the path to the the directory containing the test case generated with the j-POST test generator.

- Use directly the jar TestEngine.jar providing your desired options. Type

  java -jar TestEngine.jar -help

  for the description of the available options.

3 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 toolchain to check some properties on a travel agency application [15], 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".

3.1 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 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. The test module edition in j-POST is represented on the Fig. 2. 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 $missionCreation()$ 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).

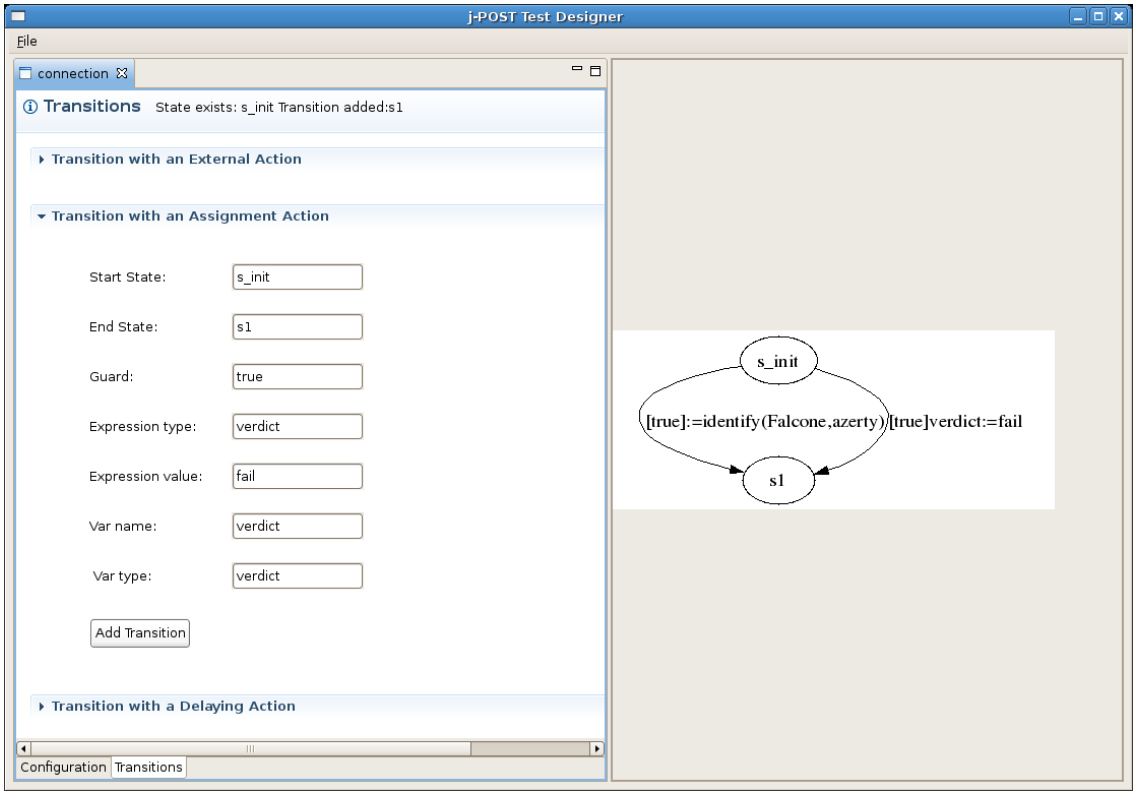

Figure 2: Edition of a test module with the designer

Possible test modules are proposed in Fig. 3, 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, 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, the j-POST test designer facilitates their writing and edition.

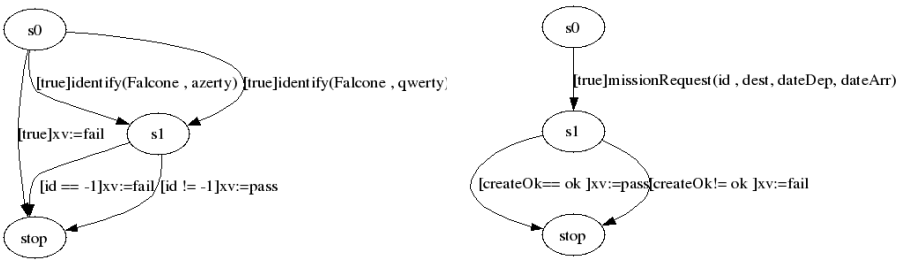

Figure 3: Test modules for predicates $connection()$ and $createMission()$

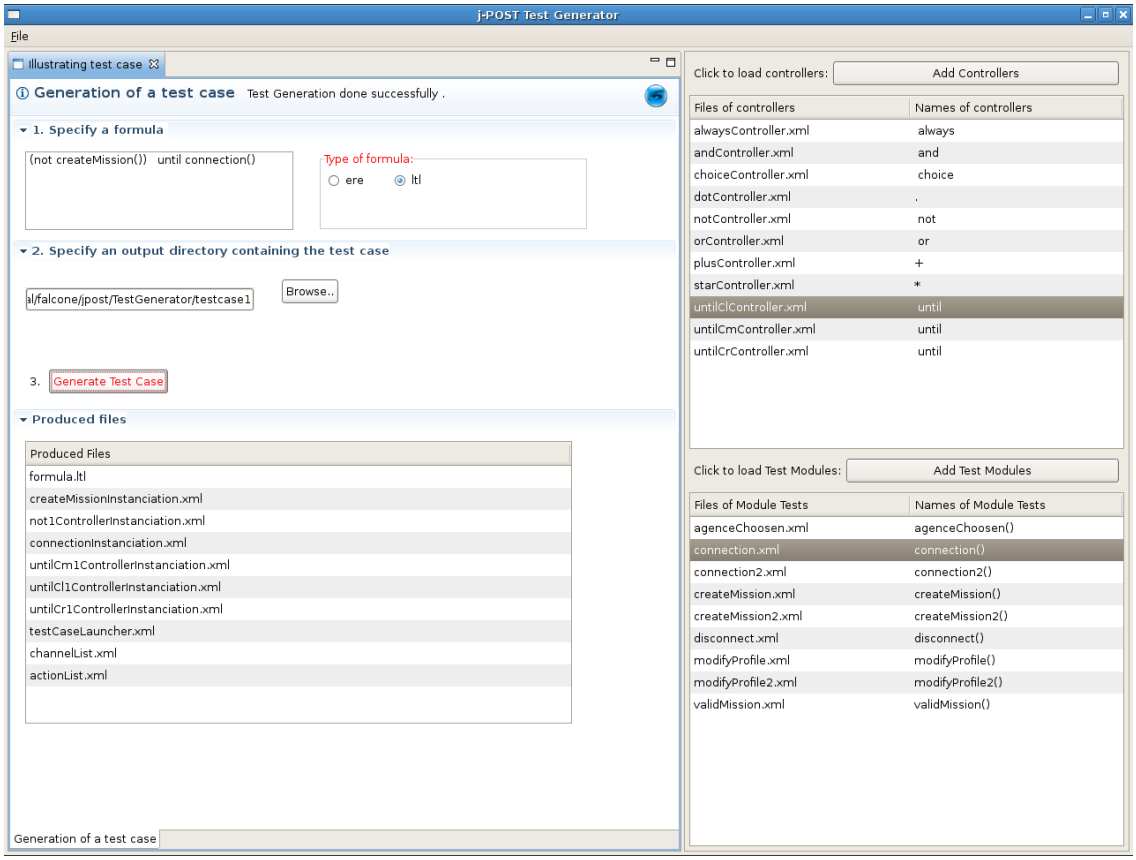

Figure 4: Generation of a test case with the test generator

3.2 Test generation

The requirement stated, and the test modules designed (Fig. 3), we are now able to perform the test generation. On Fig. 4, the generation is illustrated by a screenshot of the test generator. In order to illustrate such a generation process, we give an insight of the generated test case on Fig. 5. 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.

3.3 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*.

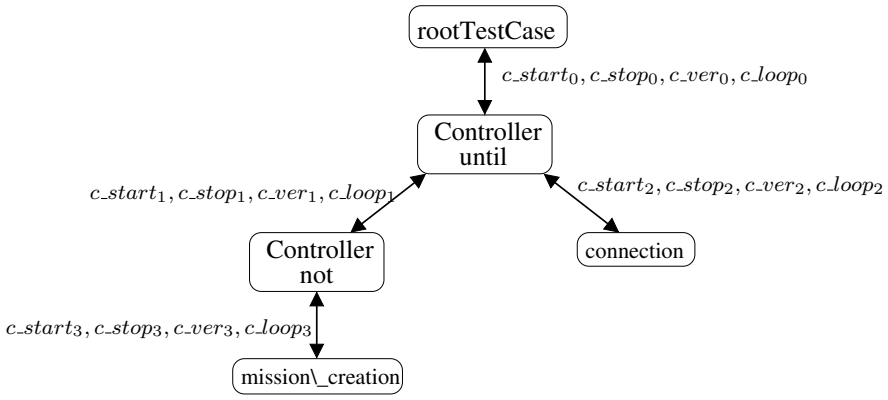

Figure 5: Test case produced from $\neg(missionCreation)\,\mathcal{U}\,connection$

Such a test objective can be obtained from the test modules given on Fig. 3. 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).

The test objective defined, and the test case generated, one can use the test execution engine to execute the test case against the SUT. Such an operation is illustrated on Fig. 6.

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.

4 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. 7).

4.1 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 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

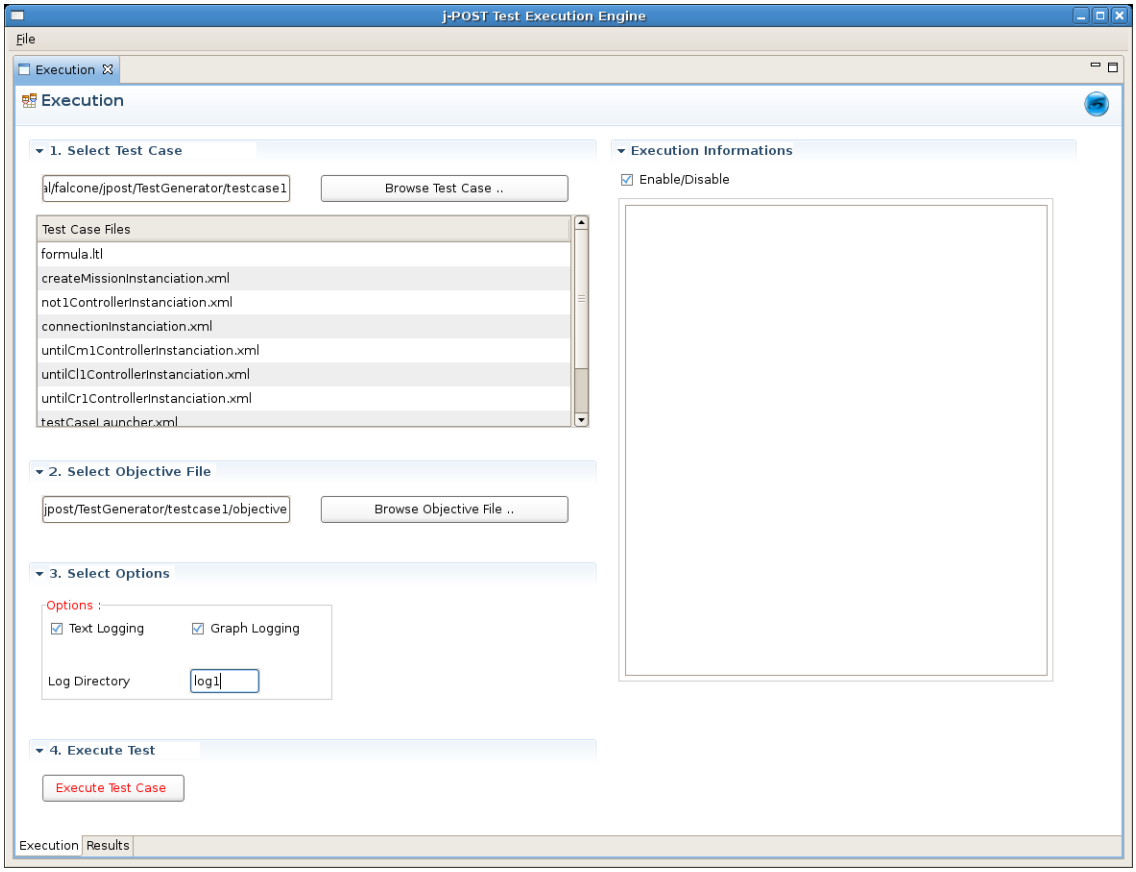

Figure 6: Execution of a test case with the test engine

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.

5 Test Generation

This section describes the functioning of the test generator and the test generation process (Fig. 8).

5.1 Overview

The j-POST test generator consists mainly in a test generation function. This function takes as input a formal requirement ϕ, 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 ϕ, an instance of the corresponding abstract test component t_i is produced ;

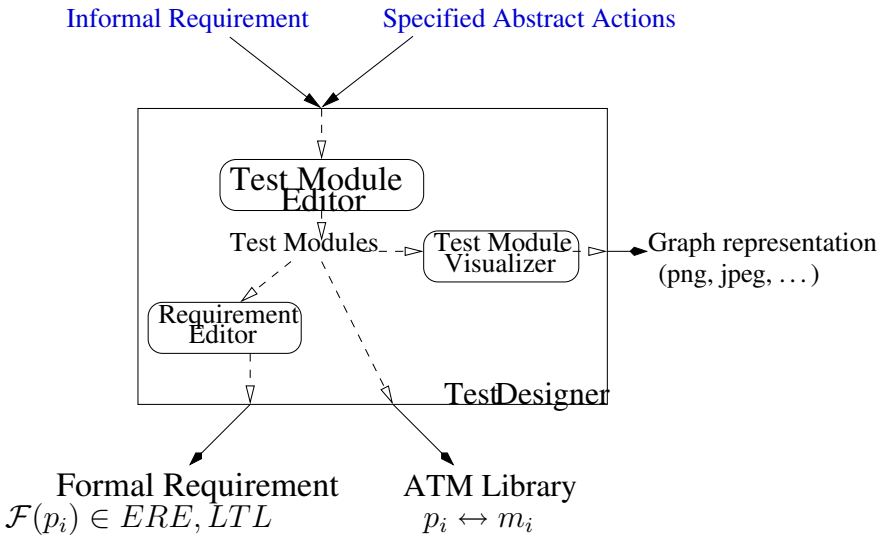

Figure 7: Architecture of the test designer of j-POST

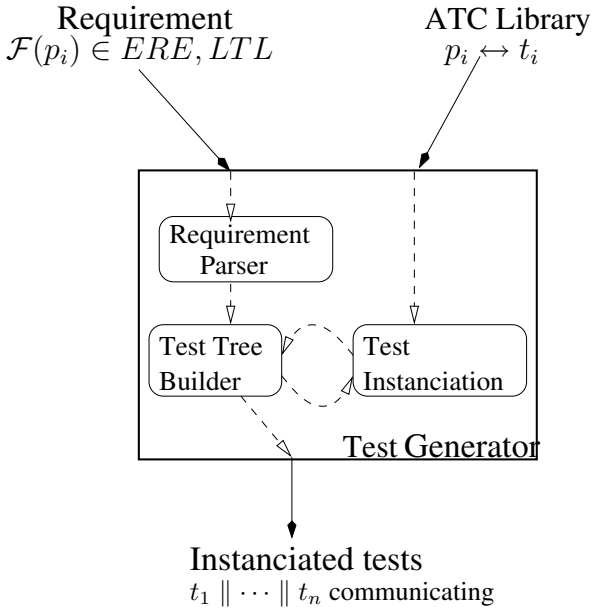

Figure 8: Architecture of the test generator of j-POST

- For each subformula $\phi = F(\phi_1, \cdots, \phi_n)$ of ϕ, 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 ϕ_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 ϕ.

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 [16] 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 [17] 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.

5.2 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 [18]. 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.

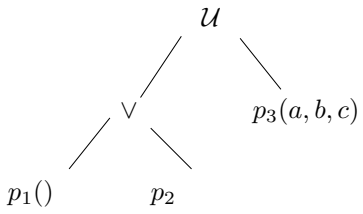

Figure 9: Abstract syntax tree corresponding to $(p_1() \vee p_2())\mathcal{U}p_3(a,b,c)$

5.3 Implementation of the test generation function (Test tree building + Test instanciation)

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. 9. 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.

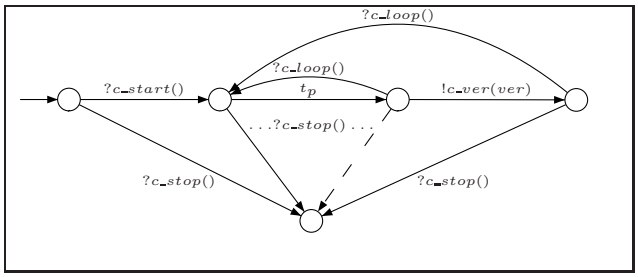

Figure 10: Instantiation of an abstract test component t_p

- FormulaNode.java: Represents the always, eventually controller of arity 1.
- PredicateNode.java: Represents the test predicate of the formula.
- BinaryOperationNode.java: Represents the controllers and, or, involve and until of arity two

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.

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 [19], 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 the generation method of this process passing in parameter a fresh canal name allowing the instantiation.
- $GT(F^n(\phi_1, \cdots, \phi_n), cs)$: Takes a channel number and generates as new numbers as the operator F has operands. Then the controller is instantiated by modifying the channel names to instantiate. After, we call successively the generation method on the children processes given them in parameter the generated channel names.
- $Test(t_p, \{c\_start, c\_stop, c\_loop, c\_ver\})$ Takes a channel number and generates control states and transitions of this predicate and implements it in the structure (see Fig. 10).

Once this stage 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. This files are also defined in a XML scheme. These two files are produced during the tree browsing by $GenTest$.

6 Test Execution

This section describes how tests are executed with the j-POST test execution engine (Fig. 11).

6.1 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*.

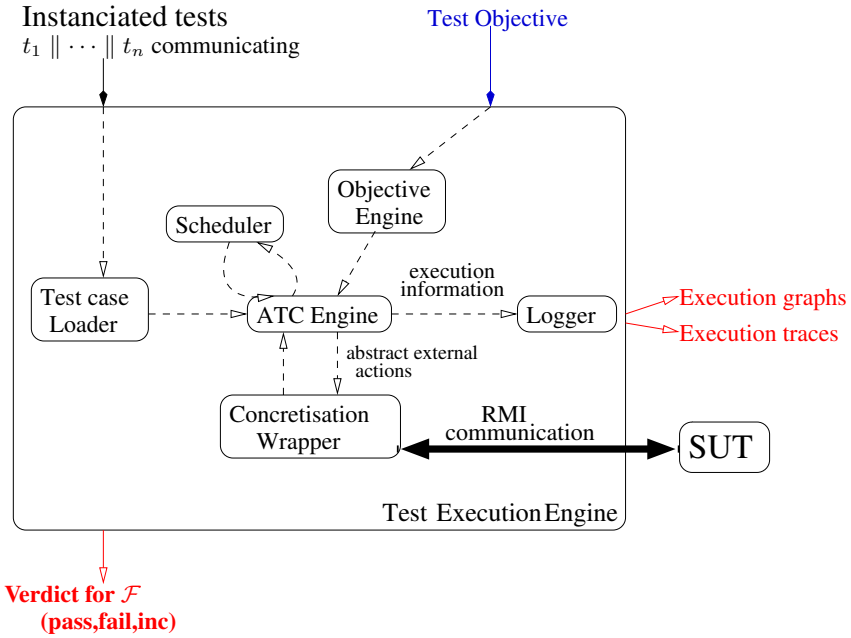

Figure 11: Architecture of the test execution engine of j-POST

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 in the abstract tests components, and and parallel composition). To bring some determinism in the test execution and to avoid inconclusive verdicts, the test execution engine uses dynamic test objectives. We distinguish two kinds of objectives. The first one are related to the verdict of the requirement formula: we select among the possible execution sequences the ones that are the more likely to lead toward a desired evaluation (depending whether we want to exhibit a correct or an incorrect behaviour of the SUT with respect to this requirement). Thus, this *global objective* (finding a valid or invalid execution sequence) is transformed into a set of *local objectives*, one for each test component. An interesting feature is that the local objectives can change *during* the test execution: for instance, to falsify the formula $\phi_1 \Leftrightarrow \phi_2$, the verdict obtained for subformula ϕ_1 may influence the local objective associated to the subformula ϕ_2. The second kind of test objective is more classical in the sense that they consist in "scenarios" (or execution patterns) the user wants to see during the test execution.

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.

6.2 Functionning 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.

6.2.1 Construction and evaluation of the test automaton

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.

6.2.2 Types of actions and execution policies

A test can perform three different kinds of actions.

- Internal. These actions are local to the process, typically data manipulations, or temporisation actions.
- Internal communication. These are synchronization operations with other processes. The process wants 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. These are actions to performed involving the SUT. It is possible to use several different parameters and wait for a returning value that can be used in the following actions of a test process. This type of action needs a synchronization stage as this action are in an abstract form. This concretisation translates the action in an action realizable on the architecture to be tested (see next subsection).

In some state a conflict can appear as soon as several actions can be performed (several guard true). The possible conflicts are:

- between internal actions,
- between external actions,
- between internal communication actions,
- between internal and external actions,
- between internal and internal communication actions,
- between external and internal communication actions,
- between all kind of actions.

6.2.3 Local policy of the test process

Several policies can be considered to favor one kind of actions or an other. In order to be 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. The scheduler has to solve conflicts when only internal communications are involved. One can easily implements several can of policy. For instance, it seems interesting to realize all execution possibilities when a conflict occurs, angle the test dynamically or change dynamically the policy during the execution.

```
<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 12: Example of how an external action is XML-coded

6.2.4 Scheduling policy

For the scheduler several policies can be also implemented. The scheduler is in charge of solve conflicts for the test processes when a choice need to be done about several internal communication actions. In the implemented policies the scheduler examines the conflicting set of actions. The process is blocked as long as a solution to the conflict is not found. Once freed, the scheduler has decided the action to be performed, the process executes it. But if there is no action, that means the process is in deadlock and no action is no longer possible.

6.3 Concretisation of external actions

The external actions are represented in an abstract form in the test processes. This abstraction allows to be independent towards the tested architecture. A concretisation phase is so 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 actions. 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.

6.3.1 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 6.3.1 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.

6.3.2 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 targetted 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.

```
<mapping>
<interface name="diag" url="rmi://distanceHost/">
<method abstractName="identify" realName="identify" arity="2" />
</interface>
</mapping>
```

- 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.

Elements of type *method* are declared for the *diag* interface. These elements correspond to the callable methods

6.4 Choice of a test objective

The classical notion of test objective has been adapted to fit in the Property-Oriented Software Testing approach. We introduced a first version of the test objectives in [22] in which we expose the motivations and the theoretical definition of test objectives.

From an abstract point of view, the test selection guided by a test objective is performed by the scheduler component of the engine. Once some test processes request to perform an operation (either a communication or an interaction with the SUT), the scheduler "follows" the test objective. The choice of the performed action is given by the semantic rule of test selection.

7 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 [23] 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 Requirement formalism grammar

B.1 Linear Temporal Logic

The initial syntax of a formula φ in LTL is given by the following grammar, where the atoms $\{p_1, \ldots, p_n\}$ are action predicates.

$$\varphi ::= \neg\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi \wedge \varphi \mid p_i$$

To be used with Java-CC we modified the grammar as follows.

```
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
```

Notice that we used the following conventions:

- the axiom is Spec
- the non-terminals start with a capital letter, the terminals are $\{;, (), \Rightarrow$ always eventualy or and not until$\}$.

And the same grammar LL(1) :

```
Formula ::= Mod Term          Dir = {always, eventually}
      |  Term          Dir = {not, name, ( }

Term ::= Term1 Term2          Dir = {name, not}

Term2 ::= OpBin Term1 Term2     Dir = {and, or, =>, until}
      | epsilon     Dir = {), ;}

Term1 ::= Factor          Dir = {name}
      | not Factor     Dir = {not}


Factor ::= Predicate           Dir = {name}
      | (Formula)      Dir = { ( }

OpBin ::= and | or | => | until
Mod ::= always | eventually

Predicate ::= name (ListParam)

ListParam ::= name EndListParam      Dir = {name}
      | epsilon          Dir = { ) }

EndListParam ::= , name          Dir = { , }
          | epsilon     Dir = { ) }
```

B.2 Extended Regular Expressions

To Be Done

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, 4.1

[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, 5.3

[4] Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools **17**(3) (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

[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

[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] j-POST Reference Page: http://www-verimag.imag.fr/jpost. (2007) 2.3

[13] The Eclipse Foundation: Eclipse Modelling Framework. http://www.eclipse.org/modeling/emf (2007) 2.3.1, 4.1

[14] AT&T Research: Graph Visualization Software. http://www.graphviz.org (2007) 2.3.1, 4.1

[15] Falcone, Y.: A Travel Agency Application. Technical report, Vérimag (2007) 3

[16] Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer-Verlag New York, Inc., New York, NY, USA (1995) 5.1

[17] 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 5.1

[18] SUN Java.net: Java-CC. http://javacc.dev.java.net (2007) 5.2

[19] Jason Hunter: JDOM. http://www.jdom.org (2007) 5.3

[20] Java Remote Method Invocation: Java RMI. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp (2007) 6.3

[21] Java Message Service: JMS. http://java.sun.com/products/jms/ (2007) 6.3

[22] 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) 6.4

[23] Project Politess: ANR-05-RNRT-01301. http://www.rnrt-politess.info (2007) 7