A Compositional Testing Framework Driven by Partial Specifications

Yliès Falcone[1], Jean-Claude Fernandez[1], Laurent Mounier[1], and Jean-Luc Richier[2]

[1] Vérimag Laboratory, 2 avenue de Vignate 38610 Gières, France
[2] LIG Laboratory, 681, rue de la Passerelle, BP 72, 38402 Saint Martin d'Hères Cedex, France
{Ylies.Falcone,Jean-Claude.Fernandez,Laurent.Mounier, Jean-Luc.Richier}@imag.fr

Abstract. We present a testing framework using a compositional approach to generate and execute test cases. Test cases are generated and combined with respect to a partial specification expressed as a set of requirements and elementary test cases. These approach and framework are supported by a prototype tool presented here. The framework is presented here in its LTL-like application, besides other specification formalisms can be added.

1 Introduction

Testing is a popular validation technique which purpose is essentially to find defects on a system implementation, either during its development, or once a final version has been completed. Therefore, and even if lots of work have already been carried out on this topic, improving the effectiveness of a testing phase while reducing its cost and time consumption remains a very important challenge, sustained by a strong industrial demand.

From a practical point of view, a test campaign consists in producing a test suite (test generation), and executing it on the target system (test execution). Automating test generation means deriving the test suite from some initial description of the system under test. The test suite consists in a set of test cases, where each test case is a set of interaction sequences to be executed by an external tester. Any execution of a test case should lead to a test verdict, indicating if the system succeeded or not on this particular test (or if the test was not conclusive).

The initial system description used to produce the test cases may be for instance the source code of the software, some hypothesis on the sets of inputs it may receive (user profiles), or some requirements on its expected properties at run-time (i.e., a characterization of its (in)-correct execution sequences). In this latter case, when the purpose of the test campaign is to check the correctness of some behavioral requirements, an interesting approach for automatic test generation is the so-called model-based testing technique. Model-based testing

A. Petrenko et al. (Eds.): TestCom/FATES 2007, LNCS 4581, pp. 107–122, 2007.

is rather successful in the communication protocol area, especially because it is able to cope with some non-determinism of the system under test. It has been implemented in several tools, see for example [1] for a survey. However, it suffers from some drawbacks that may prevent its use in other application areas. First of all, it strongly relies on the availability of a system specification, which is not always the case in practice. Moreover, when it exists, this specification should be complete enough to ensure some relevance of the test suite produced. Finally, it is likely the case that this specification cannot encompass all the implementation details, and is restricted to a given abstraction level. Therefore, to become executable, the test cases produced have to be *refined* into more concrete interaction sequences. Automating this process in the general case is still a challenging problem [2], and most of the time, when performed by hand, the soundness of the result cannot be fully guaranteed.

We propose here an alternative approach to produce a test suite dedicated to the validation of behavioral requirements of a software (see Fig. 1). In this framework the requirements $\mathcal{R}$ are expressed by logical formulas φ built upon a set of (abstract) predicates P_i describing (possibly non-atomic) operations performed on the system under test. A typical example of such requirements could be for instance a security policy, where the abstract predicates would denote some high-level operations like "user A is authenticated", or "message M has been corrupted". The approach we propose relies on the following consideration: a perfect knowledge of the implementation details is required to produce elementary test cases Tc_i able to decide whether such predicates hold or not at some state of the software execution. Therefore, writing the test cases dedicated to these predicates should be left to the programmer (or tester) expertise when a detailed system specification is not available. However, correctly orchestrating the execution of these "basic test cases" and combining their results to deduce the validity of the overall logical formula is much easier to automate since it depends only of the semantics of the operators used in this formula. This step can therefore be produced by an automatic test generator, and this test generation can even be performed in a compositional way (on the structure of the logical formula). More precisely, from the formula φ, a test generation function automatically produces an (abstract) tester AT_φ. This tester consists of a set of communicating *test controllers*, one for each operator appearing in φ. Thus, AT_φ depends only on the structure of formula φ. AT_φ is then instantiated using the elementary test cases Tc_i to obtain a concrete tester T_φ for the formula φ. Execution of this tester on the implementation I produces the final verdict.

We believe that this approach is general enough to be instantiated with several logic formalisms commonly used to express requirements on execution traces (e.g., extended regular expressions or linear temporal logics).

This works extends some preliminary descriptions on this technique [3,4] in several directions: first we try to demonstrate that it is general enough to support several logical formalisms, then we apply it for the well-known LTL temporal logic, and finally we evaluate it on a small case study using a prototype tool under development.

In addition to the numerous works proposed in the context of model-based test generation for conformance testing, this work also takes credits from the community of run-time verification. In fact, one of the techniques commonly used in this area consists in generating a monitor able to check the correctness of an execution trace with respect to a given logical requirement (see for instance [5,6] or [7] for a short survey). In practice, this technique needs to *instrument* the software under verification with a set of observation points to produce the traces to be verified by the monitor. This instrumentation should of course be correlated with the requirement to verify (i.e., the trace produced should contain enough information). In the approach proposed here, these instrumentation directives are replaced by the elementary test cases associated to each elementary predicates. The main difference is that these test cases are not restricted to pure observation actions, but they may also contain some active testing operations, like calling some methods, or communicating with some remote process to check the correctness of an abstract predicate.

The rest of the paper is organized as follows: Sect. 2 introduces the general approach, while Sect. 3 details its sound-proved application for a particular variant of the linear temporal logic LTL. Section 4 and 5 respectively describe the architecture of a prototype tool based on this framework, and its application on a small case study. The conclusion and perspectives of this work are given is Sect. 6.

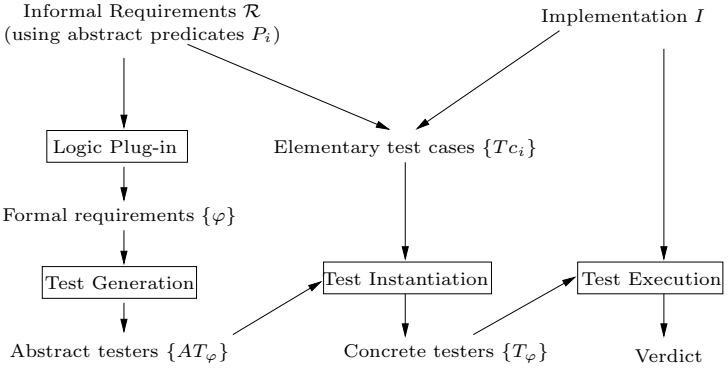

Fig. 1. Test generation overview

2 The General Approach

We describe here more formally the test generation approach sketched in the introduction. As it has been explained, this approach relies on the following steps:

- generation of an *abstract tester* AT_φ from a formal requirement φ;
- instantiation of AT_φ into a concrete tester A_φ using the set of elementary testers associated to each atomic predicate of φ;
- execution of T_φ against the System Under Test (SUT) to obtain a test verdict.

2.1 Notations

A *labelled transition system* (LTS, for short) is a quadruplet $S = (Q, A, T, q_0)$ where Q is a set of states, A a set of labels, $T \subseteq Q \times A \times Q$ the transition relation and $q_0 \in Q$ the initial state. We will denote by $p \xrightarrow{a}_T q$ (or simply $p \xrightarrow{a} q$) when $(p, a, q) \in T$. A *finite execution sequence* of S is a sequence $(p_i, a_i, q_i)_{\{0 \leq i \leq m\}}$ where $p_0 = q_0$ and $p_{i+1} = q_i$. For each finite execution sequence λ, the sequence of actions $(a_0, a_1, \ldots, a_m)$ is called a *finite execution trace* of S. We denote by $Exec(S)$ the set of all finite execution traces of S. For an execution trace $\sigma = (a_0, a_1, \ldots, a_m)$, we denote by $\mid \sigma \mid$ the length $m+1$ of σ, by $\sigma^{k \ldots l}$ the sub-sequence $(a_k, \ldots, a_l)$ when $0 \leq k \leq l \leq m$, and by $\sigma^{k \cdots}$ the sub-sequence $(a_k, \ldots, a_m)$ when $0 \leq k \leq m$. Finally, $\sigma_{\downarrow_X}$ denotes the *projection* of σ on action set X. Namely, $\sigma_{\downarrow_X} = \{a_0 \cdots a_m \mid \forall i \cdot a_i \in X \wedge \sigma = w_0 \cdot a_0 \cdots w_m \cdot a_m \cdot w_{m+1} \wedge w_i \in (A \setminus X)^*\}$.

2.2 Formal Requirements

We assume in the following that the formal requirements φ we consider are expressed using a logic $\mathscr{L}$. Formulas of $\mathscr{L}$ are built upon a finite set of *n-ary operators* F^n and a finite set of *abstract predicates* $\{p_1, p_2, \ldots, p_n\}$ as follows:

$$\text{formula} ::= F^n(\text{formula}_1, \text{formula}_2, \ldots, \text{formula}_n) \mid p_i$$

We suppose that each formula of $\mathscr{L}$ is interpreted over a finite execution trace of a LTS S, and we say that S satisfies φ (we note $S \models \varphi$) iff *all* sequences of $Exec(S)$ satisfy φ. Relation $\models$ is supposed to be defined inductively on the syntax of $\mathscr{L}$ in the usual way: abstract predicates are interpreted over $Exec(S)$, and the semantics of each operator $F^n(\varphi_1, \ldots, \varphi_n)$ is defined in terms of sets of execution traces satisfying respectively φ_1, ..., φ_n.

2.3 Test Process Algebra

In order to outline the compositionality of our test generation technique, we express a tester using an algebraic notation. We recall here the dedicated "test process algebra" introduced in [4], but other existing process algebras could also be used.

Syntax. Let *Act* be a set of *actions*, $\mathcal{T}$ be a set of *types* (with $\tau \in \mathcal{T}$), *Var* a set of *variables* (with $x \in Var$), and *Val* a set of *values* (union of values of types $\mathcal{T}$). We denote by $expr_\tau$ (resp. x_τ) any expression (resp. variable) of type τ. In particular, we assume the existence of a special type called *Verdict* which associated values are $\{pass, fail, inconc\}$ and which is used to denote the *verdicts* produced during the test execution.The syntax of a test process t is given by the following grammar:

$$\begin{aligned} t &::= [b]\,\gamma \circ t \mid t + t \mid nil \mid recX\ t \mid X \\ b &::= \text{true} \mid \text{false} \mid b \vee b \mid b \wedge b \mid \neg b \mid expr_\tau = expr_\tau \\ \gamma &::= x_\tau := expr_\tau \mid !c(expr_\tau) \mid ?c(x_\tau) \end{aligned}$$

In this grammar t denotes a basic tester (nil being the empty tester doing nothing), b a boolean expression, c a channel name, γ an action, $\circ$ is the *prefixing* operator, $+$ the *choice* operator, X a term variable, $recX$ allows *recursive* process definition (with X a term variable)[1]. When the condition b is true, we abbreviate $[true]\gamma$ by γ. Atomic actions performed by a basic tester are either internal assignments ($x_\tau := expr_\tau$), value emissions ($!c(expr_\tau)$) or value receptions ($?c(x_\tau)$) over a channel c[2].

Semantics. We first give a semantics of basic testers (t) using rewriting rule between uninterpreted terms in a CCS-like style (see Fig. 2).

$$\frac{\gamma \in Act}{[b]\gamma \circ t \xrightarrow{[b]\gamma} t}\ (\circ) \qquad \frac{t[recX \circ t/X] \xrightarrow{[b]\gamma} t' \qquad \gamma \in Act}{recX \circ t \xrightarrow{[b]\gamma} t'}\ (rec)$$

$$\frac{\gamma \in Act \qquad t_1 \xrightarrow{[b]\gamma} t'_1}{t_1 + t_2 \xrightarrow{[b]\gamma} t'_1}\ (+)_l \qquad \frac{\gamma \in Act \qquad t_2 \xrightarrow{[b]\gamma} t'_2}{t_1 + t_2 \xrightarrow{[b]\gamma} t'_2}\ (+)_r$$

Fig. 2. Rules for term rewriting

The semantics of a basic test process t is then given by means of a LTS $S_t = (Q^t, A^t, T^t, q_0^t)$ in the usual way: states Q^t are "configurations" of the form (t, ρ), where t is a term and $\rho : Var \to Val$ is an *environment*. States and transition of S_t (relation $\longrightarrow$) are the smallest sets defined by the rules given in Fig. 3 (using the auxiliary relation $\rightharpoonup$ defined in Fig. 2). The initial state q_0^t of S is the configuration (t_0, ρ_0), where ρ_0 maps all the variables to an undefined value. Finally, note that actions A^t of S_t are labelled either by internal assignments ($x_\tau := v$) or external emission ($!c(v)$). In the following we denote by $A^t_{\text{ext}} \subseteq A^t$ the external emissions and receptions performed by the LTS associated to a test process t.

Complex testers are obtained by parallel composition of test processes with synchronisation on a channel set cs (operator $\|_{cs}$), or using a so-called "join-exception" operator ($\ltimes^{\mathcal{I}}$), allowing to interrupt a process on reception of a communication using the interruption channel set $\mathcal{I}$. We note $\|$ for $\|_\emptyset$ and Act_chan(s) all possible actions using a channel in the set s. To tackle with communication in our semantics, we give two sets of rules specifying how LTSs are composed relatively to the communication operators ($\|_{cs}, \ltimes^{\mathcal{I}}$). These rules aim to maintain asynchronous execution, communication by *rendez-vous*. Let $S_i^t = (Q_i^t, A_i^t, T_i^t, q_{0i}^t)$ be two LTSs modelling the behaviours of two processes t_1 and t_2, we define the LTS $S = (Q, A, T, q_0)$ modelling the behaviours of $S_1^t \|_{cs} S_2^t$

[1] We will only consider *ground* terms: each occurrence of X is bound to $recX$.

[2] To simplify the calculus, we supposed that all channels exchange one value. In the testers, we also use "synchronisation channels", without exchanged argument, as a straightforward extension.

$$\frac{\rho(expr_\tau) = v \qquad t \xrightarrow{[b]x_\tau := expr_\tau} t' \qquad \rho(b) = true}{(t,\rho) \xrightarrow{x_\tau := v} (t', \rho[v/x_\tau])} \ (:=)$$

$$\frac{\rho(expr_\tau) = v \qquad t \xrightarrow{[b]!c(expr_\tau)} t' \qquad \rho(b) = true}{(t,\rho) \xrightarrow{!c(v)} (t', \rho)} \ (!)$$

$$\frac{v \in Dom(\tau) \qquad t \xrightarrow{[b]?c(x_\tau)} t' \qquad \rho(b) = true}{(t,\rho) \xrightarrow{!c(v)} (t, \rho[v/x_\tau])} \ (?)$$

Fig. 3. Rules for environment modification

$$\frac{p_1 \xrightarrow{a} p_1' \qquad a \notin \text{Act\_chan}(cs)}{(p_1,p_2) \xrightarrow{a} (p_1',p_2)} \ (\overline{\|^{\text{l}}_{cs}}) \qquad \frac{p_2 \xrightarrow{a} p_2' \qquad a \notin \text{Act\_chan}(cs)}{(p_1,p_2) \xrightarrow{a} (p_1,p_2')} \ (\overline{\|^{\text{r}}_{cs}})$$

$$\frac{p_1 \xrightarrow{a} p_1' \qquad p_2 \xrightarrow{a} p_2' \qquad a \in \text{Act\_chan}(cs)}{(p_1,p_2) \xrightarrow{a} (p_1',p_2')} \ (\|_{cs})$$

$$\frac{p_1 \xrightarrow{a} p_1' \qquad a \notin \text{Act\_chan}(\mathcal{I})}{(p_1,p_2) \xrightarrow{a} (p_1',p_2)} \ (\overline{\ltimes^{\mathcal{I}}}) \qquad \frac{p_2 \xrightarrow{a} p_2' \qquad a \in \text{Act\_chan}(\mathcal{I})}{(p_1,p_2) \xrightarrow{a} (\bot,p_2')} \ (\ltimes^{\mathcal{I}})$$

Fig. 4. LTS composition related to $\|_{cs}$ and $\ltimes^{\mathcal{I}}$

and $S_1^t \ltimes^{\mathcal{I}} S_2^t$ as the product of S_1^t and S_2^t where $Q \subseteq (Q_1^t \cup \{\bot\}) \times Q_2^t$ and the transition rules are given in Fig. 4.

2.4 Test Generation

Principle. The test generation technique we propose aims to produce a tester process t_φ associated to a formal requirement φ and it can be formalized by a function called *GenTest* in the rest of the paper ($GenTest(\varphi) = t_\varphi$). This generation step depends of course of the logical formalism under consideration, but it is compositionally defined in the following way:

- a basic tester t_{p_i} is associated with each abstract predicate p_i of φ;
- for each sub-formula $\phi = F^n(\phi_1, \cdots, \phi_n)$ of φ, a test process t_ϕ is produced, where t_ϕ is a parallel composition between test processes $t_{\phi_1}, \ldots, t_{\phi_n}$ and a test process $\mathtt{C}_{F^n}$ called a *test controller* for operator F^n.

The purpose of test controllers $\mathtt{C}_{F^n}$ is both to schedule the test execution of the t_{ϕ_k} (starting, stopping or restarting their execution), and to combine their verdicts to produce the overall verdict associated to ϕ. As a result, the architecture of a tester t_φ matches the abstract syntax tree corresponding to formula φ: leaves are basic tester processes corresponding to abstract predicates p_i of φ, intermediate nodes are controllers associated with operators of φ.

Hypothesis. To allow interactions between the internal sub-processes of a tester t_φ, we assume the following hypotheses:

Each tester sub-process t_{ϕ_k} (basic tester or controller) owns a special variable used to store its *local verdict*. This variable is supposed to be set to one of these values when the test execution terminates – its intuitive meaning is similar to the conformance testing case:

- *pass* means that the test execution of t_{ϕ_k} did not reveal any violation of the sub-formula associated to t_{ϕ_k};
- *fail* means that the test execution of t_{ϕ_k} did reveal a violation of the sub-formula associated to t_{ϕ_k};
- *inconc* (inconclusive) means that the test execution of t_{ϕ_k} did not allow to conclude about the validity of the sub-formula associated to t_{ϕ_k}.

Each tester process t_{ϕ_k} (basic tester or controller) owns a set of four dedicated communication channels $cs_k = \{c\_start_k, c\_stop_k, c\_loop_k, c\_ver_k\}$ used respectively to start its execution, to stop it, to resume it from its initial state and to deliver a verdict. In the following, we denote by $\mathtt{C}(cs, cs_1, \cdots, cs_n)$ each controller $\mathtt{C}$ where cs is the channel set dedicated to the communication with the embracing controller whereas the (cs_i) are the channel sets dedicated to the communication with the sub-test processes. Finally, a "starter" process is also required to start the topmost controller associated to t and to read the verdict it delivered.

Each basic tester process t_{p_i} associated to an LTS $S_{t_{p_i}}$ is supposed to have a subset of actions $A^{t_{p_i}}_{\text{ext}} \subseteq A^{t_{p_i}}$ used to communicate with the SUT. Considering $t = GenTest(\varphi)$, the set A^t_{ext} is defined as the union of the $A^{t_{p_i}}_{\text{ext}}$ where p_i is a basic predicate of φ.

Test generation function definition (GenTest). $GenTest$ can then be defined as follows using GT as an intermediate function:

$$GenTest(\varphi) \stackrel{\textbf{def}}{=} GT(\varphi, cs) \parallel_{\{c\_start, c\_ver\}} (!c\_start() \circ ?c\_ver(x) \circ nil)$$

where cs is the set $\{c\_start, c\_stop, c\_loop, c\_ver\}$ of channel names associated to t_φ.

$$GT(p_i, cs) \stackrel{\textbf{def}}{=} Test(t_{p_i}, cs)$$

$$GT(F^n(\phi_1, \ldots, \phi_n), cs) \stackrel{\textbf{def}}{=} (GT(\phi_1, cs_1) \parallel \cdots \parallel GT(\phi_n, cs_n)) \parallel_{cs'} \mathtt{C}_{F^n}(cs, cs_1, \ldots, cs_n)$$

where $cs_1, \ldots, cs_n$ are sets of fresh channel names and $cs' = cs_1 \cup \cdots \cup cs_n$.

$$Test(t_p, \{c\_start, c\_stop, c\_loop, c\_ver\}) \stackrel{\textbf{def}}{=}$$

$$recX\ (?c\_start() \circ t_p \circ !c\_ver(ver) \circ ?c\_loop() \circ X) \ltimes^{\{c\_stop\}} (?c\_stop() \circ nil)$$

2.5 Test Execution and Test Verdicts

As seen in the previous subsections, the semantics of a tester represented by a test process t is expressed by a LTS $S_t = (Q^t, A^t, T^t, q^t_0)$ where $A^t_{\text{ext}} \subseteq A^t$ denotes the external actions it may perform. Although the system under test I is not described by a formal model, its behaviour can also be expressed by a LTS $S_I = (Q^I, A^I, T^I, q^I_0)$. A *test execution* is a sequence of interactions (on

A^t_{ext}) between t and I in order to deliver a *verdict* indicating whether the test succeeded or not. We define here more precisely these notions of test execution and test verdict.

Formally speaking, a test execution of a test process t on a SUT I can be viewed as an execution trace of the parallel product $\otimes_{A^t_{\text{ext}}}$ between LTSs S_t and S_I with synchronizations on actions of A^t_{ext}. This product is defined as follows:

$S_t \otimes_{A^t_{\text{ext}}} S_I$ is the LTS (Q, A, T, q_0) where $Q \subseteq Q^t \times Q^I$, $A \subseteq A^t \cup A^I$, $q_0 = (q^t_0, q^I_0)$, and
$T = \{(p^t, p^I) \xrightarrow{a} (q^t, q^I) \mid (p^t, a, q^t) \in T^t \wedge (p^I, a, q^I) \in T^I \wedge a \in A^t_{\text{ext}}\} \cup \{(p^t, p^I) \xrightarrow{a} (q^t, p^I) \mid (p^t, a, q^t) \in T^t \wedge a \in A^t \setminus A^t_{\text{ext}}\} \cup \{(p^t, p^I) \xrightarrow{a} (p^t, q^I) \mid (p^I, a, q^I) \in T^I \wedge a \in A^I \setminus A^t_{\text{ext}}\}$.

For any test execution $\sigma \in \text{Exec}(S_t \otimes_{A^t_{\text{ext}}} S_I)$, we define the verdict function: $\text{VExec}(\sigma) = \mathit{pass}$ (resp. $\mathit{fail}, \mathit{inconc}$) iff $\sigma = c\_start() \cdot \sigma' \cdot c\_ver(pass)$ (resp. $\sigma = c\_start() \cdot \sigma' \cdot c\_ver(fail)$, $\sigma = c\_start() \cdot \sigma' \cdot c\_ver(inconc)$) and c_start (resp. c_ver) is the starting (resp. the verdict) channel associated to the topmost controller of t.

3 Application to Variant of LTL

This section presents an instantiation of the previous framework for a (non atomic) action-based version of LTL-X, the next-free variant of LTL [8].

3.1 The Logic

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

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

Semantics. Formulas φ are interpreted over the finite execution traces $\sigma \in A^*$ of a LTS. We introduce the following notations.

To each atomic predicate p_i of φ we associate a subset of actions A_{p_i} and two subsets L_{p_i} and $L_{\overline{p_i}}$ of $A^*_{p_i}$. Intuitively, A_{p_i} denotes the actions that influence the truth value of p_i, and L_{p_i} (resp. $L_{\overline{p_i}}$) the set of finite execution traces satisfying (resp. non satisfying) p_i. We suppose that the action sets A_{p_i} are such that $\{(A_{p_i})_i\}$ forms a partition of A, that for all i, j, $L_{p_i} \cap L_{\overline{p_i}} = \emptyset$ and $(L_{p_i} \cup L_{\overline{p_i}}) \cap (L_{p_j} \cup L_{\overline{p_j}}) = \emptyset$. The sets of actions for a predicate are easily extended to sets of actions for a formula: $A_{\neg\varphi} = A_\varphi$, $A_{\varphi_1 \wedge \varphi_2} = A_{\varphi_1 \mathcal{U} \varphi_2} = A_{\varphi_1} \cup A_{\varphi_2}$.

The truth value of a formula is given in a three-valued logic matching our notion of test verdicts: a formula φ can be evaluated to *true* on a trace σ ($\sigma \models_T \varphi$), or it can be evaluated to *false* ($\sigma \models_F \varphi$), or its evaluation may remain inconclusive ($\sigma \models_I \varphi$).

The semantics for a formula φ is defined by three sets. The set of sequences that satisfy (resp. violate) the formula φ is noted $[\![\varphi]\!]^T$ (resp. $[\![\varphi]\!]^F$). We also note $[\![\varphi]\!]^I$ the set of sequences for which the satisfaction remains inconclusive.

- $[\![p_i]\!]^T = \{\omega \mid \exists\, \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow_{A_{p_i}}} \in L_{p_i}\}$
  $[\![p_i]\!]^F = \{\omega \mid \exists\, \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow_{A_{p_i}}} \in L_{\overline{p_i}}\}$
- $[\![\neg\varphi]\!]^T = [\![\varphi]\!]^F$
  $[\![\neg\varphi]\!]^F = [\![\varphi]\!]^T$
- $[\![\varphi_1 \wedge \varphi_2]\!]^T = \{\omega \mid \exists\, \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow_{A_{\varphi_1}}} \in [\![\varphi_1]\!]^T \wedge \omega'_{\downarrow_{A_{\varphi_2}}} \in [\![\varphi_2]\!]^T\}$
  $[\![\varphi_1 \wedge \varphi_2]\!]^F = \{\omega \mid \exists\, \omega', \omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow_{A_{\varphi_1}}} \in [\![\varphi_1]\!]^F \vee \omega'_{\downarrow_{A_{\varphi_2}}} \in [\![\varphi_2]\!]^F\}$
- $[\![\varphi_1 \mathcal{U} \varphi_2]\!]^T = \{\omega \mid \exists\, \omega_1, \ldots, \omega_n, \omega' \cdot \omega = \omega_1 \cdots \omega_n \cdot \omega'$
  $\wedge \forall i < n \cdot \omega_{i\downarrow_{A_{\varphi_1}}} \in [\![\varphi_1]\!]^T \wedge \omega_{n\downarrow_{A_{\varphi_2}}} \in [\![\varphi_2]\!]^T\}$
  $[\![\varphi_1 \mathcal{U} \varphi_2]\!]^F = \{\omega \mid \exists\, \omega_1, \ldots, \omega_n, \omega' \cdot \omega = \omega_1 \cdots \omega_n \cdot \omega'$
  $\wedge \Big(\forall i \leq n \cdot \omega_{i\downarrow_{A_{\varphi_2}}} \in [\![\varphi_2]\!]^F \vee (\exists l \leq n \cdot \omega_{l\downarrow_{A_{\varphi_2}}} \in [\![\varphi_2]\!]^T \wedge \exists k < l \cdot \omega_{k\downarrow_{A_{\varphi_1}}} \in [\![\varphi_1]\!]^F)\Big)\}$
- $[\![\varphi]\!]^I = A^* \setminus ([\![\varphi]\!]^P \cup [\![\varphi]\!]^F)$

Finally we note $\sigma \models_T \varphi$ (resp. $\sigma \models_F \varphi$, $\sigma \models_I \varphi$) for $\sigma \in [\![\varphi]\!]^T$ (resp. $\sigma \in [\![\varphi]\!]^F$,$\sigma \in [\![\varphi]\!]^I$).

3.2 Test Generation

Following the structural test generation principle given in Sect. 2.4, it is possible to obtain a *GenTest* function for our LTL-like logic. The *GenTest* definition can be made explicit simply by giving controller definitions. So, we give a graphical description of each controller used by *GenTest*. To simplify the presentation, the *stop* transitions are not represented: the receptions all lead from each state of the controller to some "sink" state corresponding to the nil process, and emissions are sent by controllers to stop sub-tests when their execution is not needed anymore for the verdict computation.

The $\mathsf{C}_\neg(\{c\_start, c\_loop, c\_ver\}, \{c\_start', c\_loop', c\_ver'\})$ controller is shown on Fig. 5. It inverts the verdict received by transforming *pass* verdict into *fail* verdict (and conversely) and keeping *inconc* verdict unchanged.

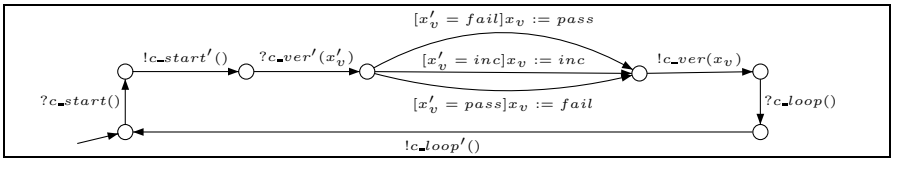

Fig. 5. The $\mathsf{C}_\neg$ controller

The $\mathsf{C}_\wedge(\{c\_start, c\_loop, c\_ver\}, \{c\_start_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_loop_r, c\_ver_r\})$ controller is shown on Fig. 6. It starts both controlled sub-tests and waits for their verdict returns, and sets the global verdict depending on received values.

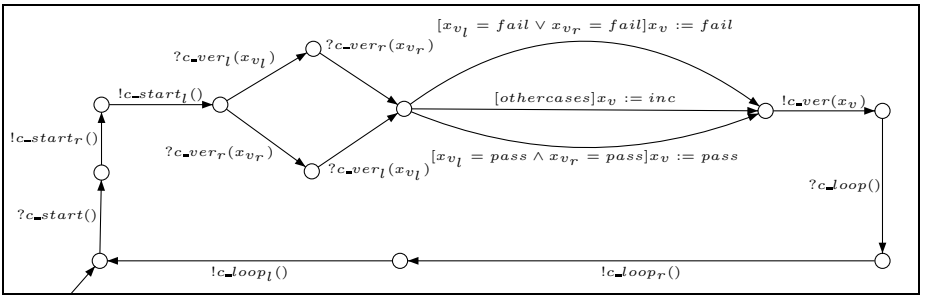

Fig. 6. The $\mathsf{C}_\wedge$ controller

The $\mathsf{C}_{\mathcal{U}}(\{c\_start, c\_loop, c\_ver\}, \{c\_start_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_loop_r, c\_ver_r\})$ controller is shown on Fig. 7 and Fig. 8. It is composed of three sub-processes executing in parallel and starting on the same action $?c\_start()$. The first sub-process C_m is represented on Fig. 7. The second and third ones corresponds to two instantiations

$$\mathsf{C}_l(\{c\_start, c\_loop, c\_ver\}, \{c\_start_l, c\_loop_l, c\_ver_l\}),$$
$$\mathsf{C}_r(\{c\_start, c\_loop, c\_ver\}, \{c\_start_r, c\_loop_r, c\_ver_r\})$$

of $\mathsf{C}_x(\{c\_start, c\_loop, c\_ver\}, \{c\_start_x, c\_loop_x, c\_ver_x\})$ for the two controlled sub-test for the two sub-formulas. An algebraic expression of this controller could be

$$\mathsf{C}_{\mathcal{U}}(\cdots) = (\mathsf{C}_l(\cdots) \parallel \mathsf{C}_r(\cdots)) \parallel_{\{r\_fail, l\_fail, r\_pass, l\_pass\}} \mathsf{C}_m(\cdots)$$

One could understand C_l and C_r as two sub-controllers in charge of communicating with the controlled tests that send relevant information to the "main" sub-controller C_m deciding the verdict. The reception of an inconclusive verdict from a sub-test process interrupts the controller which emits an inconclusive verdict (not represented on the figure). If no answer is received from the sub-processes after some finite amount of time, then the tester delivers its verdict (*timeout* transitions). For the sake of clarity we simplified the controller representation. First, we represent the emission of the controller verdict and the return to the initial state under a reception of a loop signal ($?c\_loop()$) by a state which name represents the value of the emitted verdict. Second, we do not represent *inconc* verdict, the controller propagates it.

3.3 Soundness Proposition

We express that an abstract test case produced by the *GenTest* function is always *sound*, *i.e.* it delivers a *pass* (resp. *fail*) verdict when it is executed on a SUT behavior I only if the formula used to generate it is satisfied (resp. violated) on I. This proposition relies on one hypothesis, and two intermediate lemmas.

Hypothesis 1. *Each test case t_{p_i} associated to a predicate p_i is strongly sound in the following sense:*

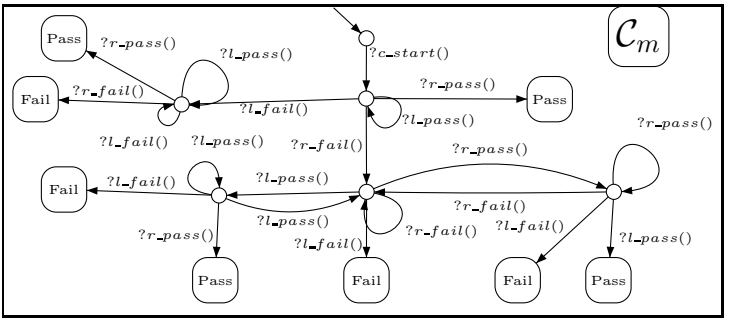

Fig. 7. The $\mathsf{C}_{\mathcal{U}}$ controller, the C_m part

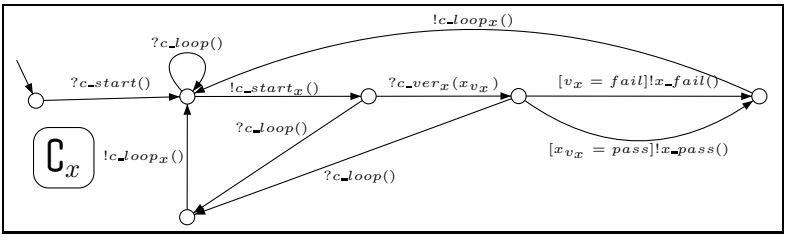

Fig. 8. The $\mathsf{C}_{\mathcal{U}}$ controller, the C_x part

$$\forall \sigma \in \mathrm{Exec}(t_{p_i} \otimes_{A_{p_i}} I), \mathrm{VExec}(\sigma) = pass \Rightarrow \sigma \models_T p_i$$
$$\forall \sigma \in \mathrm{Exec}(t_{p_i} \otimes_{A_{p_i}} I), \mathrm{VExec}(\sigma) = fail \Rightarrow \sigma \models_F p_i$$

The lemmas state that the verdict computed by t_φ on a sequence σ only depends on actions of σ belonging to A_φ.

Lemma 1. *All execution sequences with the same projection on a formula φ actions have the same satisfaction relation towards φ. That is:*

$$\forall \sigma, \sigma' \cdot \sigma_{\downarrow A_\varphi} = \sigma'_{\downarrow A_\varphi} \Rightarrow (\sigma \models_T \varphi \Leftrightarrow \sigma' \models_T \varphi) \wedge (\sigma \models_F \varphi \Leftrightarrow \sigma' \models_F \varphi)$$

Lemma 2. *For each formula φ, each sequence σ, the verdicts pass and fail of a sequence do not change if we project it on φ's actions. That is:*

$$\forall \varphi, \forall \sigma \cdot \sigma \models_T \varphi \Rightarrow \sigma_{\downarrow A_\varphi} \models_T \varphi$$
$$\forall \varphi, \forall \sigma \cdot \sigma \models_F \varphi \Rightarrow \sigma_{\downarrow A_\varphi} \models_F \varphi$$

These lemmas come directly from the definition of our logic and the controllers used in *GenTest*. Now we can formulate the proposition.

Theorem 1. *Let φ be a formula, and t=GenTest(φ), S a LTS, $\sigma \in Exec(t \otimes_{A_\varphi} S)$ a test execution sequence, the proposition is:*

$$VExec(\sigma) = pass \Longrightarrow \sigma \models_T \varphi$$
$$VExec(\sigma) = fail \Longrightarrow \sigma \models_F \varphi$$

Sketch of the soundness proof. The proof is done by structural induction on φ. We give the proof for two cases.

For the predicates. The proof relies directly on predicate strong soundness (Hypothesis 1).

For the negation operator. Let suppose $\varphi = \neg\varphi'$. We have to prove that:

$$\forall \sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I), \text{VExec}(\sigma) = pass \Rightarrow \sigma \models_T \neg\varphi'$$
$$\forall \sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I), \text{VExec}(\sigma) = fail \Rightarrow \sigma \models_F \neg\varphi'$$

Let $\sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I)$ suppose that $\text{VExec}(\sigma) = pass$.
By definition of GT,

$$GT(\neg\varphi', \mathcal{L}) = GT(\varphi', \mathcal{L}') \parallel_{\mathcal{L}'} \mathcal{C}_\neg(\mathcal{L}, \mathcal{L}')$$

Since controller $\mathcal{C}_\neg$ does not trigger the *c_loop* transition of its subtest when it is used as a main tester process, execution sequence σ is necessarily in the form:

$$c\_start() \cdot \sigma_I \cdot \sigma' \cdot \sigma_I \cdot (([x_v = pass]x_{v_g} := fail \mid [x_v = fail]x_{v_g} := pass \mid [x_v = inconc]x_{v_g} := inconc) \cdot \sigma_I \cdot c\_ver(x_{v_g})$$

with $\sigma' \in \text{Exec}(GT(\varphi', \mathcal{L}') \otimes_{A_{\varphi'}} I)$, σ_I denoting SUT's actions, and $\omega \cdot (a \mid b) \cdot \omega'$ denoting the sequences $\omega \cdot a \cdot \omega'$ and $\omega \cdot b \cdot \omega'$.

As the controller emits a *pass* verdict ($!c\_ver(x_{v_g})$ with x_{v_g} evaluated to *pass* in the $\mathcal{C}_\neg$'s environment) it means that it necessarily received a *fail* verdict ($[x_v = fail]x_{v_g} := pass$) on *c_ver'* from the sub-test corresponding to $GT(\varphi', \mathcal{L}')$. So we have $\sigma' \in \text{Exec}(GT(\varphi', \mathcal{L}') \otimes_{A'_\varphi} I)$ and $\text{VExec}(\sigma') = fail$.

The induction hypothesis implies that $\sigma' \models_F \varphi'$. The Lemma 2 gives that ${\sigma'}_{\downarrow_{A_{\varphi'}}} \models_F \varphi'$. And we have:

$$\begin{aligned} {\sigma'}_{\downarrow_{A_{\varphi'}}} &= {\sigma'}_{\downarrow_{A_\varphi}} \ (\forall \varphi, A_\varphi = A_{\neg\varphi}) \\ &= \sigma_{\downarrow_{A_\varphi}} \ (c\_start, \sigma_I \notin {A_\varphi}^*) \end{aligned}$$

So $\sigma_{\downarrow_{A_\varphi}} \models_F \varphi'$. We conclude using the Lemma 1 that $\sigma \models_F \varphi'$ that is $\sigma \models_T \neg\varphi'$. The proof for $\forall \sigma \in \text{Exec}(GT(\neg\varphi', \mathcal{L}) \otimes_{A_\varphi} I), \text{VExec}(\sigma) = fail \Rightarrow \sigma \models_F \neg\varphi'$ is similar.

Others operators. Proofs for the other operators follow the same principle and can be found in [9].

4 Java-CTPS

We now present Java-CTPS, a prototype of a testing framework tool for the Java environment which follows our approach. We just describe an abstract view of the tool. Interested readers can refer to [9] which contains a more detailed description.

Java-CTPS contains a test generator using the compositional approach for Java, *i.e.* the tester is generated in Java, for a SUT written in the same language. An interface is provided for the user to write a library of elementary test cases from the SUT interface. Indeed, the interface defines a set of methods that can be called. Elementary test cases are terms of our test calculus which external actions correspond to these methods: execution of an external action on the tester leads to a complete execution of the method on the SUT (from the call to the return). An elementary test case execution on the tester leads to the execution of some methods in the SUT interface.

Afterwards, using our method, the tool transforms a specification in a given formalism in a abstract test case. Then it is combined with the library to provide an executable test case.

Synthesis algorithms of controlled tests for different formalisms have been defined and implemented. Two interfaces are provided to the user: a command-line mode and a graphic interface. A simplified version of the test generation and execution is depicted on Fig. 9.

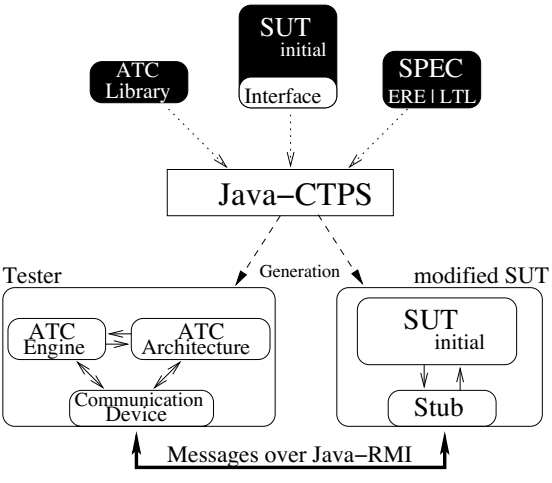

Fig. 9. Simplified working principle

Elementary test cases library establishment. The SUT's architecture description provides the set of controllable and observable actions on the system. The user can compose them, with respect to the test calculus, and write new elementary test cases. Programming is eased by the abstraction furnished by the SUT interface.

Specification as a set of requirements. Java-CTPS offers several formalisms to express requirements on the system.

- *Temporal logics.* Temporal logics [8] are frequently used to express specification for reactive systems. Their use has proved to be useful and appreciated

for system verification. Our case studies have shown that many concepts in security policies are in the scope of these logics.
- *Regular Expression.* Regular expressions [10] allows to define behaviour schemes expressed on a system traces. They are commonly used and well-understood by engineers for their practical aspect.

Test of a system. Java-CTPS translates the specification into abstract test cases following the specification formula structure. Depending on the used specification formalism and the expressed requirement, the tool generates a test architecture whose test cases are coming from the controller library in accordance with *Gen-Test*. An execution engine is also generated. So, the generated tester can execute different test cases translated into a unique representation formalism on the test calculus engine. The initial SUT is also modified by adding a stub to communicate with the tester. This component provides means to launch method calls on the reception of specific signals. Thus, abstract test cases executing on the tester guide concrete test cases execution on the modified SUT. Communication between tester and SUT is done using the Java-RMI mechanism as we plan to support distributed SUT in a future evolution of our tool.

5 Case Study

We present a case study illustrating the approach presented above. From some credit card security documents [11], we established a security policy and a credit card model. We applied our method with the security policy as a partial specification and the executable credit card model as a SUT. The credit card model and part of its security policy are overviewed here.

The card. The architecture of the credit card is presented on Fig. 10. The interface is modeled by the *Device* component, corresponding to the possible action set on the card. Several banking operations are proposed, *e.g. provide_pin*, *change_pin*, *init_session*, *transaction*, *close_session*. Choice was made to use a Java interface to model the banking application one. The *Device* component interacts with a *Memory Abstraction* component providing, as its name indicates, some basic operations on the memory's areas. The *Memory* is just the credit card memory represented as a fixed size integer array.

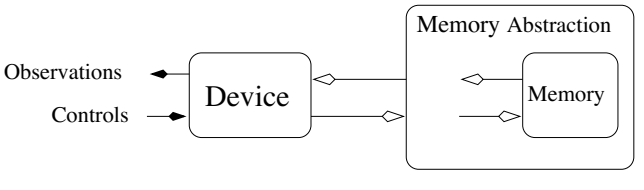

Fig. 10. The credit card architecture

The security policy. Our study allowed us to extract several security requirement specific to the credit card security domain. These requirements concerned several specification formalisms: regular expressions, and temporal logics. Some examples of properties that we were able to test can be expressed at an informal level:

1. After three failed authentications, the card is blocked, i.e. no action is permitted anymore.
2. If the card is suddenly removed the number of remaining authentications tries is set to 0.
3. No action is permitted without an identification.

For example one could see the first property formalised in our logic as several rules, one for each possible action:

$$try\_3\_authentications(all\_failed) \Longrightarrow action(blocked)$$

The third one could be reasonably understood as:

$$action(blocked)\ \mathcal{U}\ authentication(success)$$

With this formalisation, these properties were tested with test cases that use elementary combinations of card interface actions. For example we wrote an abstract test case leading to three failed authentications using actions *provide_pin* and *init_session*.

6 Conclusion

In this work we have proposed a testing framework allowing to produce and execute test cases from a partial specification of the system under test. The approach we follow consists in generating the test cases from some high-level requirements on the expected system behaviour (expressed in a trace-based temporal logic), assuming that a concrete elementary tester is provided for each abstract predicate used in these requirements. This "partial specification" plays a similar role to the instrumentation directives currently used in run-time verification techniques, and we believe that they are easier to obtain in a realistic context than a complete operational specification. Furthermore, we have illustrated how this approach could be instantiated on a particular logic (an action-based variant of LTL-X), while showing that it is general enough to be applied to other similar trace-based logics. Finally, a prototype tool implementing this framework is available and preliminary experiments have been performed on a small case study.

Our main objective is now to extend this prototype in order to deal with larger examples. A promising direction is to investigate how the so-called MOP technology [6] could be used as an implementation platform. In particular, it already offers useful facilities to translate high-level requirements (expressed in various logics) into (passive) observers, and to monitor the behaviour of a program under

test using these monitors. A possible extension would then be to replace these observers by our active basic testers (using the aspect programming techniques supported by MOP).

Acknowledgement. The authors thank the referees for their helpful remarks.

References

1. Hartman, A.: Model based test generation tools survey. Technical report, AGEDIS Consortium (2002)
2. van der Bijl, M., Rensink, A., Tretmans, J.: Action refinement in conformance testing. In: Khendek, F., Dssouli, R. (eds.) Testing of Communicating Systems (TESTCOM) LNCS, vol. 3205, pp. 81–96. Springer, Heidelberg (2005)
3. Darmaillacq, V., Fernandez, J.C., Groz, R., Mounier, L., Richier, J.L.: Test generation for network security rules. In: TestCom, pp. 341–356 (2006)
4. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A test calculus framework applied to network security policies. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification. LNCS, vol. 4262, pp. 55–69. Springer, Heidelberg (2006)
5. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
6. Chen, F., D'Amorim, M., Roşu, G.: Checking and correcting behaviors of java programs at runtime with java-mop. In: Workshop on Runtime Verification (RV'05), ENTCS, vol. 144(4), pp. 3–20 (2005)
7. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. Theor. Comput. Sci. 336(2-3), 209–234 (2005)
8. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. New York, Inc. Springer, Heidelberg (1995)
9. Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A partial specification driven compositional testing method and tool. Technical Report TR-2007-04, Vérimag Research Report (2007)
10. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Shannon, C.E., McCarthy, J. (eds.) Automata Studies, pp. 3–41. Princeton University Press, Princeton, New Jersey (1956)
11. Mantel, H., Stephan, W., Ullmann, M., Vogt, R.: Guideline for the development and evaluation of formal security policy models in the scope of itsec and common criteria. Technical report, BSI,DFKI (2004)