A Partial Specification Driven Compositional Testing Method

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

Verimag Research Report n^o TR-2007-4

February 2007

A Partial Specification Driven Compositional Testing Method

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

February 2007

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 and regular expressions applications, besides other specification formalisms can be added.

Keywords: composition, partial-specification, requirement, test, formalism, concurrency, calculus

Notes:

How to cite this report:

@techreport { ,
title = { A Partial Specification Driven Compositional Testing Method},
authors = { Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, Jean-Luc Richier},
institution = { Verimag Research Report },
number = {TR-2007-4},
year = { 2007},
note = { }
}

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. As such, it is still a major step of software validation process. 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 of producing a test suite (test generation), and executing it on the target system (test execution). Automating both phases helps reducing the overall cost.

Automating test generation means deriving the test suite from some initial description of the system under test. The test suite consists of 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. It consists of selecting some execution sequences (or execution trees) from an operational model of the expected behavior of the system under test (called a system specification). The selection criteria can then be driven by some coverage directives or test purposes, depending on the requirements to be tested. Finally, this execution tree is turned into a test case by adding some verdicts indicating whether these requirements have been falsified or not during the test execution.

This approach happens to be rather successful in the communication protocol area, especially because it is able to cope with some non-determinism of the system under test. A particular instance has been standardized by the telecommunication authority, and several tools implement this approach (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 requirements of a software. Instead of a complete specification, we use a partial one. Each requirement is expressed by a logical formula built upon a set of (abstract) predicates 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 test cases 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). 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). It could be summarized as follows (see Fig. 1):

1. The user needs to provide three inputs: the software implementation I; a list of informal requirements $\mathcal{R}$ defined over a set of abstract predicates P_i; and a set of elementary test cases Tc_i associated to

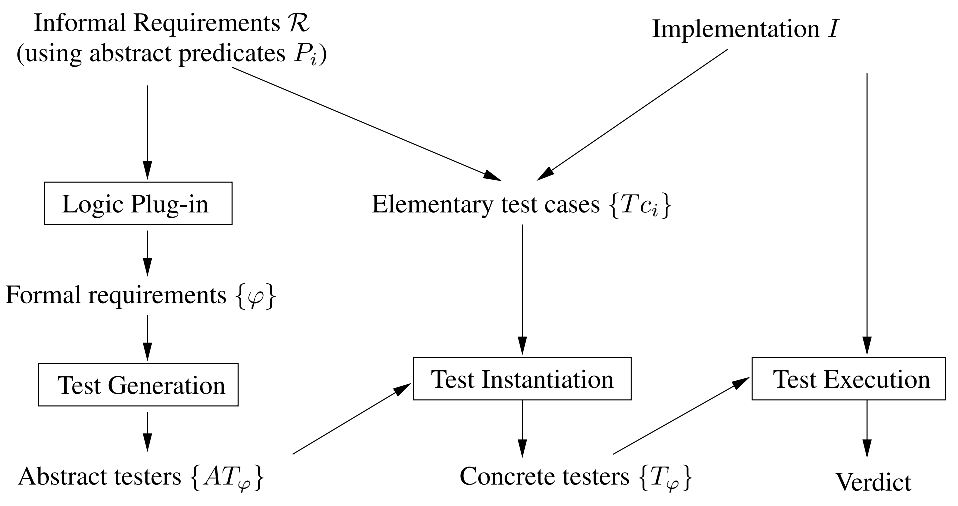
Informal Requirements $\mathcal{R}$
(using abstract predicates P_i)
Implementation I
Logic Plug-in
Elementary test cases $\{Tc_i\}$
Formal requirements $\{\varphi\}$
Test Generation
Test Instantiation
Test Execution
Abstract testers $\{AT_\varphi\}$
Concrete testers $\{T_\varphi\}$
Verdict

Figure 1: Test generation overview

each predicate P_i for the implementation I.

2. The user also needs to formalize each requirement of $\mathcal{R}$ by a logic formula φ (choosing a suitable logic formalism). φ is built upon the abstract predicates P_i and a set of operators depending on the logic considered.

3. 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.

4. Finally, AT_φ is 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.

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 regular expressions, 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 real-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 and to regular expressions. The conclusion and perspectives of this work are given is Sect. 5. Finally, the Sect. 6 gives a discussion about our approach.

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 $\sigma^{k\cdots l}$ 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, $\sigma_{\downarrow X} = \{a_0 \cdot \cdots \cdot a_m \mid \forall i \cdot a_i \in X \wedge \sigma = w_0 \cdot a_0 \cdots w_m \cdot a_m \cdot w_{m+1} \wedge w_i \in (A \setminus X)^*\}$.

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 $\varphi_1, \ldots, \varphi_n$.

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.

2.3.1 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:

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

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

$$\frac{\gamma \in Act \qquad t_1 \overset{[b]\gamma}{\rightharpoonup} t_1'}{t_1 + t_2 \overset{[b]\gamma}{\rightharpoonup} t_2'}\ (+) \qquad \frac{\gamma \in Act \qquad t_2 \overset{[b]\gamma}{\rightharpoonup} t_2'}{t_1 + t_2 \overset{[b]\gamma}{\rightharpoonup} t_2'}\ (+)$$

Figure 2: Rules for term rewriting

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

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

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

Figure 3: Rules for environment modification

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].

2.3.2 Semantics.

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

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 $\rho : Var \rightarrow Val$ 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 $\mathrm{Act\_chan}(s)$ 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 $S_1^t \|_{cs} S_2^t$ (resp. $S_1^t \ltimes^{\mathcal{I}} S_2^t$) 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 (resp. in Fig. 5).

[1] We will only consider *ground* terms: each occurrence of X is binded 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.

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

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

Figure 4: LTS composition related to $\|_{cs}$

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

Figure 5: LTS composition related to $\ltimes^{\mathcal{I}}$

2.4 Test Generation

The test generation technique we propose aims to produce a tester process $t_{\mathcal{R}}$ associated to a formal requirement $\mathcal{R}$ of a logic $\mathscr{L}$ and it can be formalized by a function called $GenTest_{\mathscr{L}}$ in the rest of the paper ($GenTest_{\mathscr{L}}(\mathcal{R}) = t_{\mathcal{R}}$).

2.4.1 Principle

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 $\mathcal{R}$;
- for each sub-requirement $r = F^n(r_1, \cdots, r_n)$ of $\mathcal{R}$, a test process t_r is produced, where t_r is a parallel composition between test processes $t_{r_1}, \ldots, t_{r_n}$ 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_{r_k} (starting, stopping or restarting their execution), and to combine their verdicts to produce the overall verdict associated to r. As a result, the architecture of a tester $t_{\mathcal{R}}$ matches the abstract syntax tree corresponding to requirement $\mathcal{R}$: leaves are basic tester processes corresponding to abstract predicates p_i of $\mathcal{R}$, intermediate nodes are controllers associated with operators of $\mathcal{R}$.

2.4.2 Hypothesis

To allow interactions between the internal sub-processes of a tester $t_{\mathcal{R}}$, we assume the following hypotheses:

- Each tester sub-process t_{r_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_{r_k} did not reveal any violation of the sub-requirement associated to t_{r_k};
  - $fail$ means that the test execution of t_{r_k} did reveal a violation of the sub-requirement associated to t_{r_k};
  - $inconc$ (inconclusive) means that the test execution of t_{r_k} did not allow to conclude about the validity of the sub-requirement associated to t_{r_k}.

- Each tester process t_{r_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 top most 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 $t_{\mathcal{R}} = GenTest_{\mathscr{L}}(\mathcal{R})$, the set $A^{t_{\mathcal{R}}}_{\text{ext}}$ is defined as the union of the $A^{t_{p_i}}_{\text{ext}}$ where p_i is a basic predicate of $\mathcal{R}$.

2.4.3 Test-generation function definition

$GenTest_{\mathscr{L}}$ can then be defined as follows, using $GT_{\mathscr{L}}$ as an intermediate function:

DEFINITION 2.1 (TEST GENERATION FUNCTION $GenTest_{\mathscr{L}}$) *The test generation function $GenTest_{\mathscr{L}}$ is defined by structural induction on its argument: a requirement $\mathcal{R}$.*

$$GenTest_{\mathscr{L}}(\mathcal{R}) \overset{def}{=} GT_{\mathscr{L}}(\mathcal{R}, cs) \parallel_{\{c\_start, c\_ver\}} (!c\_start() \circ ?c\_ver(x) \circ nil)$$

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

$$GT_{\mathscr{L}}(p_i, cs) \overset{def}{=} Test(t_{p_i}, cs)$$

$$GT_{\mathscr{L}}(F^n(\phi_1, \dots, \phi_n), cs) \overset{def}{=} (GT_{\mathscr{L}}(\phi_1, cs_1) \parallel \cdots \parallel GT_{\mathscr{L}}(\phi_n, cs_n)) \parallel_{cs'} \mathtt{C}_{F^n}(cs, cs_1, \dots, cs_n)$$

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

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

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

We now deal with the notions of test execution and test verdicts.

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:

DEFINITION 2.2 (EXECUTION OF A TEST RELATIVELY TO A SUT) *$S_t \otimes_{A^t_{\text{ext}}} S_I$ 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*
$T = \{(p^t, p^I) \xrightarrow{a} (q^t, q^I) \mid (p^t, a, q^t) \in T^t \wedge (p^I, a, q^I) \in T^I \wedge a \in A^t_{\text{ext}}\} \cup \{(p^t, p^I) \xrightarrow{a} (q^t, p^I) \mid (p^t, a, q^t) \in T^t \wedge a \in A^t \setminus A^t_{\text{ext}}\} \cup \{(p^t, p^I) \xrightarrow{a} (p^t, q^I) \mid (p^I, a, q^I) \in T^I \wedge a \in A^I \setminus A^t_{\text{ext}}\}$.

We associate to test exection a notion of verdict defined as follows:

DEFINITION 2.3 (TEST VERDICT) *For any test execution $\sigma \in \mathrm{Exec}(S_t \otimes_{A^t_{\text{ext}}} S_I)$, we define the verdict function:* $\mathrm{VExec}(\sigma) =$ *pass (resp. fail, 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 top most controller of t.*

3 Applications

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] and extended regular expressions[9].

Preliminaries. We first introduce the following notations:

- To each atomic predicate p_i 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_{LTL}: $A_{\neg\varphi} = A_\varphi, A_{\varphi_1 \wedge \varphi_2} = A_{\varphi_1} \cup A_{\varphi_2}, A_{\varphi_1 \mathcal{U} \varphi_2} = A_{\varphi_1} \cup A_{\varphi_2}$.

A_R: $A_{\neg R} = A_R, A_{R_1+R_2} = A_{R_1} \cup A_{R_2}, A_{R_1 \cdot R_2} = A_{R_1} \cup A_{R_2}, A_{R^+} = A_R$.

- The truth value of a formula φ or an expression R is given in a three-valued logic matching our notion of test verdicts: *e.g.* 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 principle is similar for a regular expression.

3.1 Application to a variant of LTL

Now we present an instantiation of the previous framework for a (non atomic) action-based version of LTL-X, the next-free variant of LTL [8]. This choice is typical in the verification community as LTL-X is insensitive to stuttering [10]. We also have a time notion not related to the sampling time of actions but more on global patterns of actions related to the predicates.

3.1.1 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.
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. $\sigma \in [\![\varphi]\!]^F$,$\sigma \in [\![\varphi]\!]^I$).

3.1.2 Test generation

Following the structural test generation principle given in Sect.2.4, it is possible to obtain a $GenTest_{\mathrm{LTL}}$ function for our LTL-like logic. The $GenTest_{\mathrm{LTL}}$ definition can be made explicit simply by giving controller definition. So, we give a graphical description of each controller used by $GenTest_{\mathrm{LTL}}$. To simplify the presentation, the *stop* transitions are represented using a graphical trick: 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 $\mathtt{C}_\neg(\{c\_start, c\_stop, c\_loop, c\_ver\}, \{c\_start', c\_stop', c\_loop', c\_ver'\})$ controller is shown on Fig. 6. It inverts the verdict received by transforming $pass$ verdict into $fail$ verdict (and conversely) and keeping $inconc$ verdict unchanged.

- The $\mathtt{C}_\wedge(\{c\_start, c\_stop, c\_loop, c\_ver\}, \{c\_start_l, c\_stop_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_stop_r, c\_loop_r, c\_ver_r\})$ controller is shown on Fig. 7. It starts both controlled sub-tests and waits for their verdict returns, and sets the global verdict depending on received values.

- The $\mathtt{C}_\mathcal{U}(\{c\_start, c\_stop, c\_loop, c\_ver\}, \{c\_start_l, c\_stop_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_stop_r, c\_loop_r, c\_ver_r\})$ controller is shown on Fig. 8 and Fig. 9. It is composed of three sub-processes executing in parallel and starting on the same action $?c\_start()$. The fist sub-process corresponds to the one represented on Fig. 8 namely $\mathtt{C}_m$. The second and third ones corresponds to two instantiations

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

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

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

.

One could understand $\mathtt{C}_l$ and $\mathtt{C}_r$ as two sub-controllers in charge of communicating with the controlled tests that send relevant information to the "main" sub-controller $\mathtt{C}_m$ 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 simplify 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.

3.1.3 Soundness proposition

We express that an abstract test case produced by the $GenTest_{LTL}$ 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 3.1 *Each test case t_{p_i} associated to a predicate p_i is strongly sound in the following sense:*

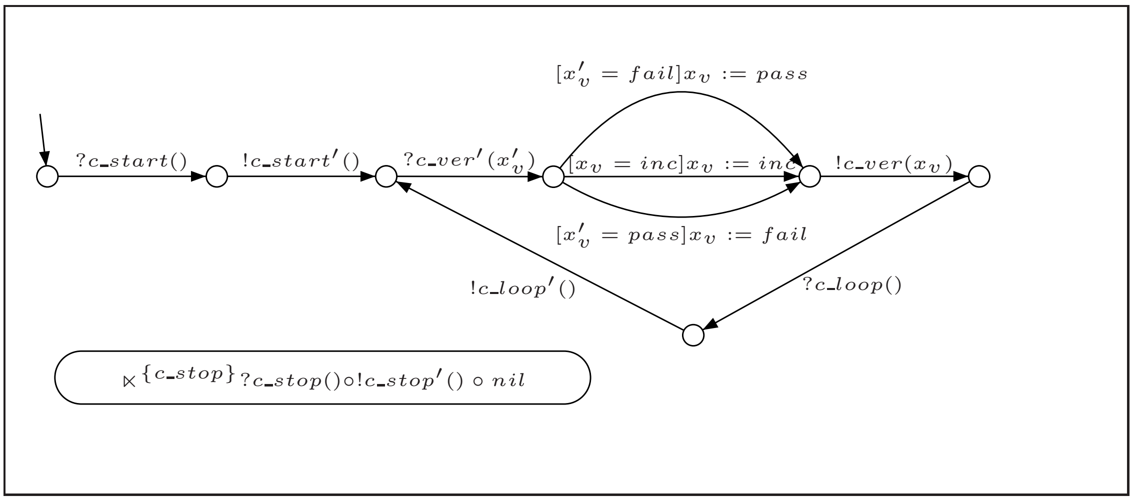
$[x'_v = fail]x_v := pass$
$?c\_start()$
$!c\_start'()$
$?c\_ver'(x'_v)$
$[x_v = inc]x_v := inc$
$!c\_ver(x_v)$
$[x'_v = pass]x_v := fail$
$!c\_loop'()$
$?c\_loop()$
$\ltimes^{\{c\_stop\}} ?c\_stop() \circ !c\_stop'() \circ nil$

Figure 6: The $\mathtt{C}_\neg$ controller

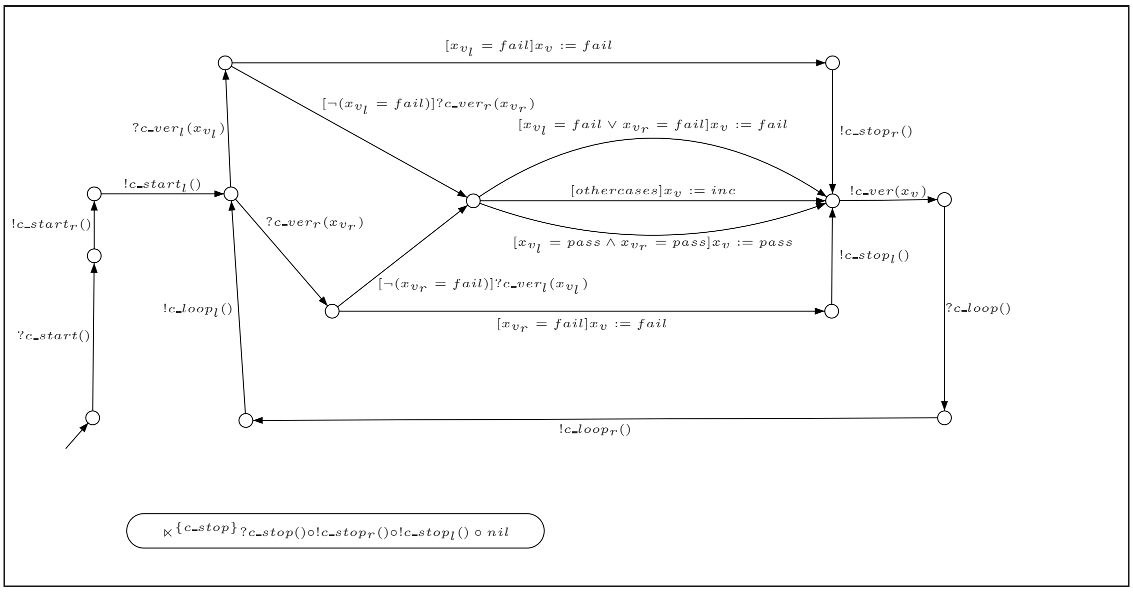
$[x_{v_l} = fail]x_v := fail$
$[\neg(x_{v_l} = fail)]?c\_ver_r(x_{v_r})$
$[x_{v_l} = fail \vee x_{v_r} = fail]x_v := fail$
$?c\_ver_l(x_{v_l})$
$!c\_stop_r()$
$!c\_start_l()$
$[othercases]x_v := inc$
$!c\_ver(x_v)$
$!c\_start_r()$
$?c\_ver_r(x_{v_r})$
$[x_{v_l} = pass \wedge x_{v_r} = pass]x_v := pass$
$!c\_stop_l()$
$[\neg(x_{v_r} = fail)]?c\_ver_l(x_{v_l})$
$!c\_loop_l()$
$?c\_loop()$
$[x_{v_r} = fail]x_v := fail$
$?c\_start()$
$!c\_loop_r()$
$\ltimes^{\{c\_stop\}} ?c\_stop() \circ !c\_stop_r() \circ !c\_stop_l() \circ nil$

Figure 7: The $\mathtt{C}_\wedge$ controller

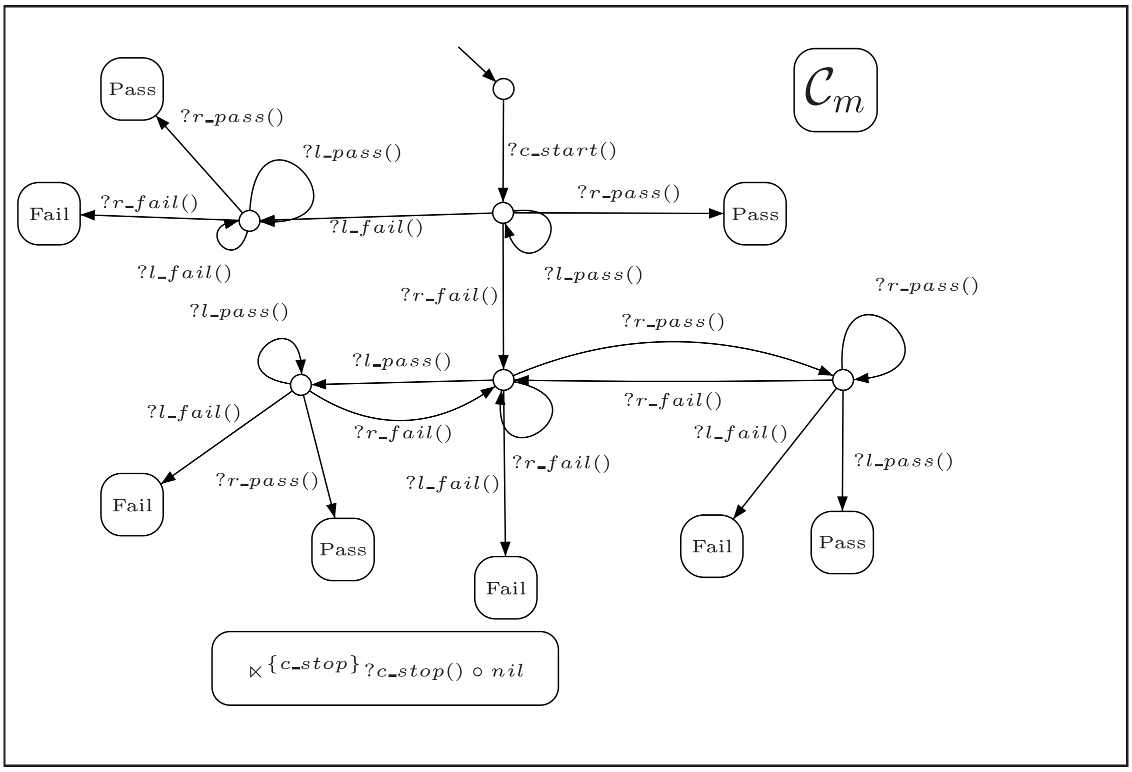

Figure 8: The $\mathsf{C}_{\mathcal{U}}$ controller, athe C_m part

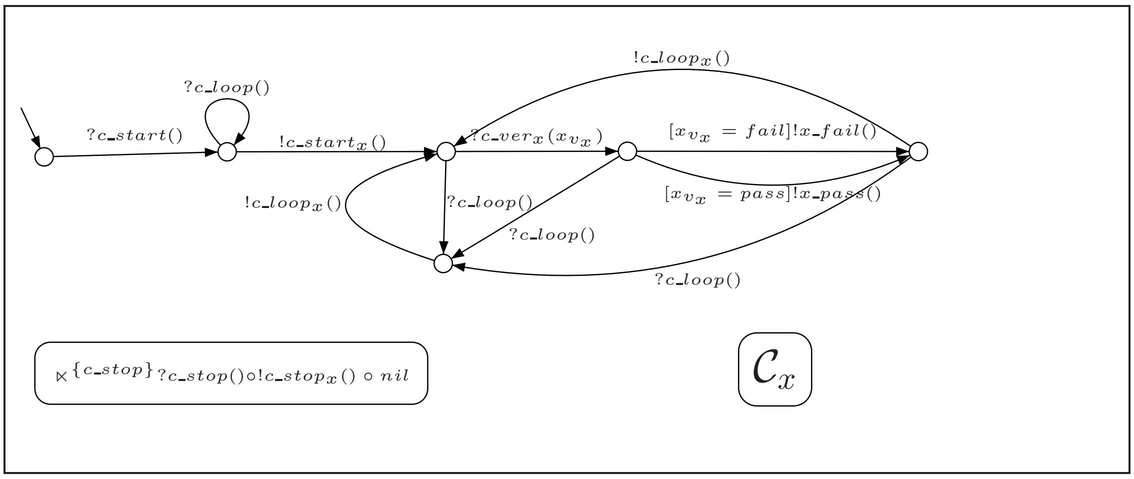

Figure 9: The $\mathsf{C}_{\mathcal{U}}$ controller, the C_x part

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

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 3.1 *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 3.2 *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 $\mathit{GenTest}_{\text{LTL}}$. Now we can formulate the proposition.

THEOREM 3.1 *Let φ be a formula, and $t = \mathit{GenTest}_{\text{LTL}}(\varphi)$, S a LTS, $\sigma \in \mathit{Exec}(t \otimes_{A_\varphi} S)$ a test execution sequence, the proposition is:*

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

Lemmas and the soundness proposition are proved in Appendix. A.

3.2 Application to Regular Expressions

This section presents an application of the framework where the requirements are expressed with some kind of regular expressions [9].

3.2.1 Regular expressions

Syntax. Original regular expression did not consider the negation. We have decided to include them in the syntax, as so one could specify undesired behavior. The syntax of a regular expression R is given by the following grammar, where the atoms $\{p_1, \ldots, p_n\}$ are action predicates.

$$R ::= \emptyset \mid p_i \mid \neg R \mid R + R \mid R^+ \mid R.R$$

Semantics. We now give the semantics of the regular expression chosen in our framework. In a same fashion as for the temporal logic used, the semantics of a regular expression is the set of action sequences it describes.

- $[\![\emptyset]\!]^T = A^*$
  $[\![\emptyset]\!]^F = \emptyset$
- $[\![p_i]\!]^T = \{\omega \mid \exists\,\omega',\omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow A_{p_i}} \in L_{p_i}\}$
  $[\![p_i]\!]^F = \{\omega \mid \exists\,\omega',\omega'' \cdot \omega = \omega' \cdot \omega'' \wedge \omega'_{\downarrow A_{p_i}} \in L_{\overline{p_i}}\}$
- $[\![\neg R]\!]^T = [\![R]\!]^F$
  $[\![\neg R]\!]^F = [\![R]\!]^T$
- $[\![R + R']\!]^T = [\![R]\!]^T \cup [\![R']\!]^T$
  $[\![R + R']\!]^F = [\![R]\!]^F \cap [\![R']\!]^F$

- $[\![R.R']\!]^T = \{\omega \mid \exists\omega' \cdot \omega''\omega = \omega' \cdot \omega'' \wedge \omega' \in [\![R]\!]^T \wedge \omega'' \in [\![R]\!]^T\}$
  $[\![R.R']\!]^F = \{\omega \mid \exists\omega' \cdot \omega''\omega = \omega' \cdot \omega'' \wedge \omega' \in [\![R]\!]^F \vee \omega'' \in [\![R]\!]^F\}$
- $[\![R^+]\!]^T = \{\omega \mid \exists n > 0 \cdot \omega = \omega_1 \cdots \omega_n \cdot \forall i \leq n \cdot w_i \in [\![R]\!]^T\}$
  $[\![R^+]\!]^F = \{\omega \mid \exists\omega', \omega'', \omega''' \cdot \omega = \omega' \cdot \omega'' \cdot \omega''' \wedge \omega'' \in [\![R]\!]^F\}$
- $[\![R]\!]^I = A^* \setminus ([\![R]\!]^T \cup [\![R]\!]^F)$

3.2.2 Test generation

Following the same principle used for the LTL-X like logic, we describe the $GenTest_{ERE}$ function by the controllers used in its expression.

- The $\mathtt{C}_\neg(\{c\_start, c\_stop, c\_loop, c\_ver\}, \{c\_start', c\_stop', c\_loop', c\_ver'\})$ controller is the same as the one for the future time logic (shown on Fig. 6). It inverts the verdict received by transforming $pass$ verdict into *fail* verdict (and conversely) and keeping $inconc$ verdict unchanged.
- The $\mathtt{C}_{\text{choice}}(\{c\_start, c\_stop, c\_loop, c\_ver\}, \{c\_start_l, c\_stop_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_stop_r, c\_loop_r, c\_ver_r\})$ controller is shown on Fig. 10. It realizes the semantics of the $+$ operator, *i.e.* the choice. It starts both controlled sub-tests and waits for their verdict returns. To decide a *pass* verdict, it needs to receive at least one *pass* verdict from one of the controlled subtest. To decide a *fail* verdict, both of the subcontrolled subtests have to respond *fail*. The *inconclusive* verdict are then decided when one of the controlled emits an *inconclusive* verdict and the other one do not respond *pass*.
- The $\mathtt{C}.(\{c\_start, c\_stop, c\_loop, c\_ver\}, \{c\_start_l, c\_stop_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_stop_r, c\_loop_r, c\_ver_r\})$ controller realizes the sequentialization of the controlled subtests (shown on Fig. 11). It starts the first controlled subtest, and waits for its verdict. If the verdict received from the first subtest is *pass*, it starts the second subtest. Otherwise, the second subtest is not started and the first subtest verdict is the verdict for the controller. The verdict of the second subtest is the verdict corresponding to the global test.
- The $\mathtt{C}_+(\{c\_start, c\_loop, c\_ver\}, \{c\_start', c\_stop', c\_loop', c\_ver'\})$ controller realizes the semantics of the $^+$ operator (shown on Fig. 12). It allows the controlled sub-test to execute several times (at least one). To decide a $pass$ verdict, all executions of the controlled test should emit a *pass* verdict. Since inn testing we consider finite execution, the controller has to decide at a time that the formula is satisfied or not. A timer mechanism is included in the controller when the controlled subtest is supposed to be finished. The intuitive meaning of the timer purpose is to allow the controller to take a decision about the verdict only considering previous received verdicts.

3.2.3 Soundness proposition

We express that an abstract test case produced by the $GenTest_{\text{ERE}}$ 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 expression used to generate it is satisfied (resp. violated) on I.
Similarly to the LTL-proposition, this one relies on one hypothesis, and two intermediate lemmas.

Hypothesis 3.2 *Each test case t_{p_i} associated to a predicate p_i is strongly sound in the following sense:*

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

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

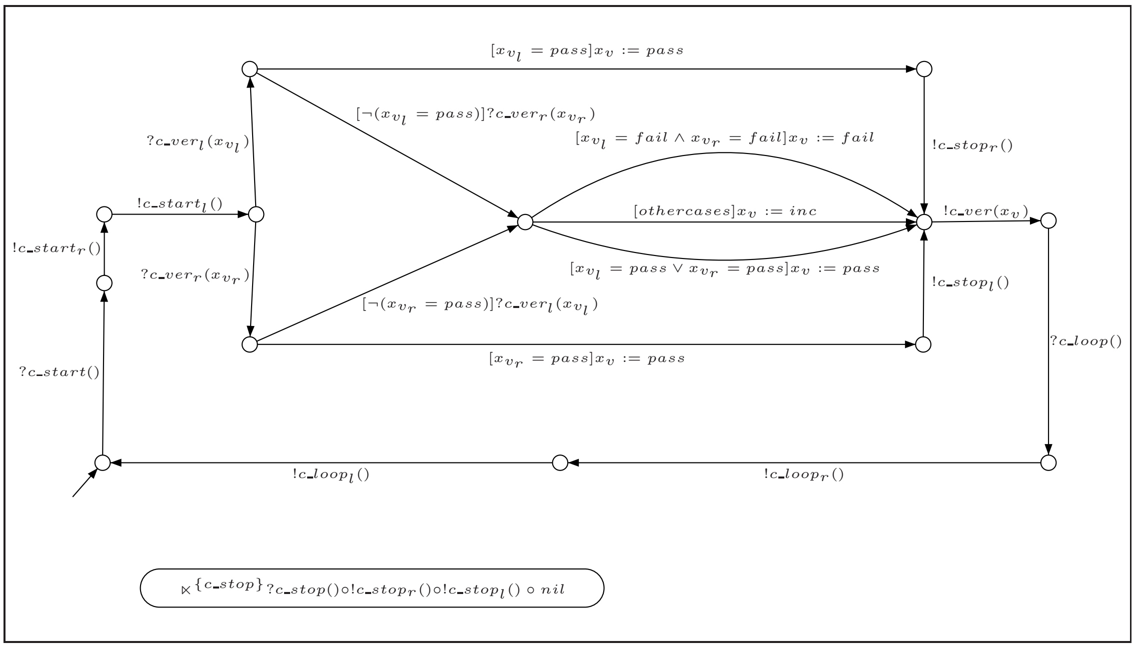
$[x_{v_l} = pass]x_v := pass$
$[\neg(x_{v_l} = pass)]?c\_ver_r(x_{v_r})$
$[x_{v_l} = fail \wedge x_{v_r} = fail]x_v := fail$
$?c\_ver_l(x_{v_l})$
$!c\_stop_r()$
$!c\_start_l()$
$[othercases]x_v := inc$
$!c\_ver(x_v)$
$!c\_start_r()$
$?c\_ver_r(x_{v_r})$
$[x_{v_l} = pass \vee x_{v_r} = pass]x_v := pass$
$!c\_stop_l()$
$[\neg(x_{v_r} = pass)]?c\_ver_l(x_{v_l})$
$?c\_loop()$
$?c\_start()$
$[x_{v_r} = pass]x_v := pass$
$!c\_loop_l()$
$!c\_loop_r()$
$\ltimes^{\{c\_stop\}}?c\_stop()\circ!c\_stop_r()\circ!c\_stop_l() \circ nil$

Figure 10: The $\mathsf{C}_{\text{choice}}$ controller

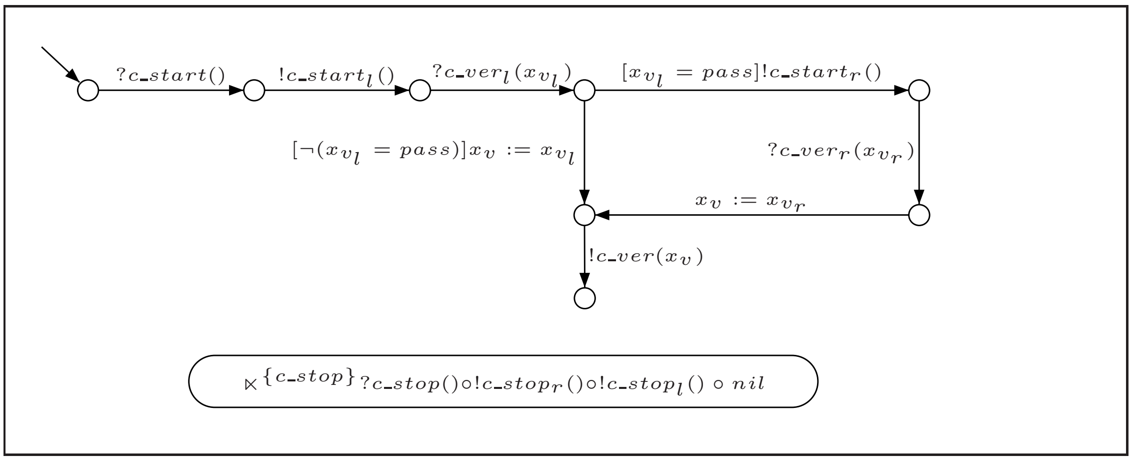
$?c\_start()$
$!c\_start_l()$
$?c\_ver_l(x_{v_l})$
$[x_{v_l} = pass]!c\_start_r()$
$[\neg(x_{v_l} = pass)]x_v := x_{v_l}$
$?c\_ver_r(x_{v_r})$
$x_v := x_{v_r}$
$!c\_ver(x_v)$
$\ltimes^{\{c\_stop\}}?c\_stop()\circ!c\_stop_r()\circ!c\_stop_l() \circ nil$

Figure 11: The $\mathsf{C}_{.}$ controller

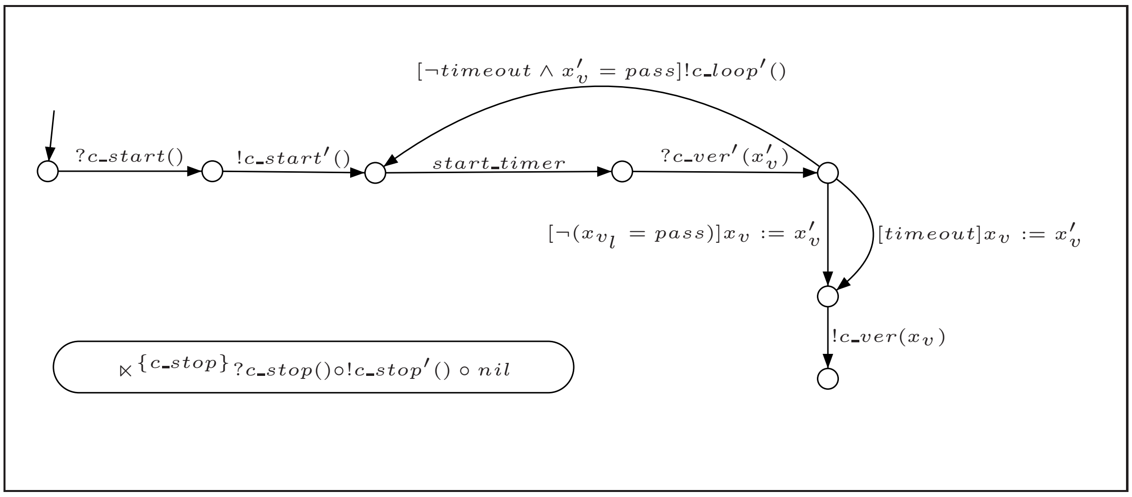
$[\neg timeout \wedge x'_v = pass]!c\_loop'()$
$?c\_start()$
$!c\_start'()$
$start\_timer$
$?c\_ver'(x'_v)$
$[\neg(x_{v_l} = pass)]x_v := x'_v$
$[timeout]x_v := x'_v$
$!c\_ver(x_v)$
$\ltimes^{\{c\_stop\}} ?c\_stop() \circ !c\_stop'() \circ nil$

Figure 12: The $\mathtt{C}_+$ controller

LEMMA 3.3 *All execution sequences with the same projection on an expression R actions have the same satisfaction relation towards R. That is:*

$$\forall \sigma, \sigma' \cdot \sigma_{\downarrow_{A_R}} = \sigma'_{\downarrow_{A_R}} \Rightarrow (\sigma \models_T R \Leftrightarrow \sigma' \models_T R) \wedge (\sigma \models_F R \Leftrightarrow \sigma' \models_F R)$$

LEMMA 3.4 *For each expression R, each sequence σ, the verdicts pass and fail of a sequence do not change if we project it on R's actions. That is:*

$$\forall R, \forall \sigma \cdot \sigma \models_T R \Rightarrow \sigma_{\downarrow_{A_R}} \models_T R$$
$$\forall R, \forall \sigma \cdot \sigma \models_F R \Rightarrow \sigma_{\downarrow_{A_R}} \models_F R$$

These lemmas come directly from the definition of our logic and the controllers used in $GenTest_{\mathrm{ERE}}$. Now we can formulate the proposition.

THEOREM 3.2 *Let R be a regular expression, and $t = GenTest_{\mathrm{ERE}}(R)$, S a LTS, $\sigma \in Exec(t \otimes_{A_R} S)$ a test execution sequence, the proposition is:*

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

4 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 instrumention 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) and the regular expressions. We believe that we have also shown that it is general enough to be applied to other similar trace-based formalism. 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 reviewers of the TESTCOM/FATES'07 conference for their helpful remarks.

5 Discussion

Implementing our approach we noticed some issues. First problem regards verdict emission and schedulling of different processes. A situation may occur where a verdict has been decided on an abstract test case and can not be emitted because the listening process is not ready to do so either because the global scheduler "forget" these processes. This leads us to maka a fairness hypothesis about the global scheduler. We believe that a prioritized verdict transmission is sufficient in most of the cases. So our implementation includes a mechanism to favour verdict propagation in controllers.

A Proofs for the LTL instantiations

This section contains the different proofs disseminated along this report.

A.1 Proof of Lemma 1

This proof is done by structural induction on φ.

A.1.1 For the predicates p_i.

Let σ, ω two sequences of A^* such that $\sigma_{\downarrow A_{p_i}} = \omega_{\downarrow A_{p_i}}$. Suppose that $\sigma \models_T p_i$. By definition that means:

$$\exists\, \sigma', \sigma'' \cdot \sigma = \sigma' \cdot \sigma'' \wedge \sigma'_{\downarrow A_{p_i}} \in L_{p_i}$$

$\sigma'_{\downarrow A_{p_i}}$ is a prefix of $\sigma_{\downarrow A_{p_i}}$, *i.e.* $\exists s \in A^* \cdot \sigma_{\downarrow A_{p_i}} = \sigma'_{\downarrow A_{p_i}} \cdot s$ And, $\omega_{\downarrow A_{p_i}} = \sigma_{\downarrow A_{p_i}}$. Then $\exists \tilde{\omega}, \tilde{\omega}' \cdot \omega_{\downarrow A_{p_i}} = \tilde{\omega} \cdot \tilde{\omega}' \wedge \tilde{\omega} = \sigma'_{\downarrow A_{p_i}}$. So, $\tilde{\omega} \in L_{p_i}$. Let $\omega^\clubsuit$ be a prefix of ω s.t. ${\omega^\clubsuit}_{\downarrow A_{p_i}} = \tilde{\omega}$. So it exists $w^\clubsuit$ s.t. $\omega = \omega^\clubsuit \cdot {\omega'}^\clubsuit$ and ${\omega^\clubsuit}_{\downarrow A_{p_i}} \in L_{p_i}$
That is $\omega \models_T p_i$. So we have $\sigma \models_T p_i \Rightarrow \omega \models_T p_i$.

Similarly we can show that $\sigma \models_F p_i \Rightarrow \omega \models_F p_i$.

A.1.2 For the $\neg$ operator.

In this case $\varphi = \neg\psi$, the induction hypothesis is on ψ, that is:

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

We consider two sequences σ, ω of A^* such that $\sigma_{\downarrow A_\varphi} = \omega_{\downarrow A_\varphi}$. Let suppose that $\sigma \models_T \varphi$, we have to show that $\omega \models_T \varphi$.
By definition of $\sigma \models_T \varphi$ we have $\sigma \models_F \psi$. And as $\forall \varphi \cdot A_\varphi = A_{\neg\varphi}$, we have $\sigma_{\downarrow A_\psi} = \omega_{\downarrow A_\psi}$. We can then apply the induction hypothesis on ψ, *i.e.* $\sigma \models_F \psi$ gives us $\omega \models_F \psi$. That is $\omega \models_T \neg\psi$.

Similarly we can show that $\sigma \models_F \varphi \Rightarrow \sigma' \models_F \varphi$

A.1.3 For the $\wedge$ operator.

In this case $\varphi = \varphi_1 \wedge \varphi_2$, the induction hypothesis is on φ_1 and φ_2, that is:

$$\forall \sigma, \sigma' \cdot \sigma_{\downarrow A_{\varphi_1}} = \sigma'_{\downarrow A_{\varphi_1}} \Rightarrow (\sigma \models_T \varphi_1 \Leftrightarrow \sigma' \models_T \varphi_1) \wedge (\sigma \models_F \varphi_1 \Leftrightarrow \sigma' \models_F \varphi_1)$$

$$\forall \sigma, \sigma' \cdot \sigma_{\downarrow A_{\varphi_2}} = \sigma'_{\downarrow A_{\varphi_2}} \Rightarrow (\sigma \models_T \varphi_2 \Leftrightarrow \sigma' \models_T \varphi_2) \wedge (\sigma \models_F \varphi_2 \Leftrightarrow \sigma' \models_F \varphi_2)$$

We consider two sequences σ, ω of A^* such that $\sigma_{\downarrow A_\varphi} = \omega_{\downarrow A_\varphi}$. Let suppose that $\sigma \models_T \varphi$, we have to show that $\omega \models_T \varphi$.
From $\sigma \models_T \varphi_1 \wedge \varphi_2$ we deduce using the definition that $\sigma \models_T \varphi_1$ and $\sigma \models_T \varphi_2$.
And $\sigma_{\downarrow A_{\varphi_1 \wedge \varphi_2}} = \omega_{\downarrow A_{\varphi_1 \wedge \varphi_2}} \Leftrightarrow \sigma_{\downarrow A_{\varphi_1} \cup A_{\varphi_2}} = \omega_{\downarrow A_{\varphi_1} \cup A_{\varphi_2}}$. We deduce that $\sigma_{\downarrow A_{\varphi_1}} = \omega_{\downarrow A_{\varphi_1}}$. We can now use the induction hypothesis twice. Then we find that: $\omega \models_T \varphi_1$. Using the same reasoning on φ_2 we find that $\omega \models_T \varphi_2$.
So we have $\omega \models_T \varphi_1$ and $\omega \models_T \varphi_2$. That is by definition $\omega \models_T \varphi_1 \wedge \varphi_2$.

Same arguments apply to show that $\sigma \models_F \varphi_1 \wedge \varphi_2 \Rightarrow \omega \models_F \varphi_1 \wedge \varphi_2$

A.1.4 For the $\mathcal{U}$ operator.

In this case $\varphi = \varphi_1 \mathcal{U} \varphi_2$ and we consider two sequences σ, ω of A^* such that $\sigma_{\downarrow A_\varphi} = \omega_{\downarrow A_\varphi}$. Let suppose that $\sigma \models_T \varphi$, we have to show that $\omega \models_T \varphi$.
By definition of $\sigma \models_T \varphi_1 \mathcal{U} \varphi_2$:

$$\exists n \in \mathbb{N} \cdot \omega = \sigma_1 \cdots \sigma_n \cdot \sigma' \wedge \forall i \leq n \cdot \sigma_{i_{\downarrow A_{\varphi_1}}} \models \varphi_1 \wedge \sigma_n \models_T \varphi_2$$

Also, from $\sigma_{\downarrow A_{\varphi_1 \mathcal{U} \varphi_2}} = \omega_{\downarrow A_{\varphi_1 \mathcal{U} \varphi_\in}}$ we deduce $\sigma_{\downarrow A_{\varphi_1 \cup \varphi_2}} = \omega_{\downarrow A_{\varphi_1 \cup \varphi_2}}$ and then $\sigma_{\downarrow A_{\varphi_1}} = \omega_{\downarrow A_{\varphi_1}}$. We can then find sub-sequences $\omega_1, \ldots, \omega_n, \omega'$ of ω such that $\omega = \omega_1 \cdots \omega_n \cdot \omega'$ and $\forall i < n \cdot \omega_{i_{\downarrow A_{\varphi_1}}} = \sigma_{i_{\downarrow A_{\varphi_1}}}$.
We can apply the induction hypothesis $n-1$ times and find $\forall i < n \cdot \omega_{i_{\downarrow A_{\varphi_1}}} \models \varphi_1$.
In a same fashion we show $\omega_n \models_T \varphi_2$.

A.2 Proof of Lemma 2

This lemma is a straightforward corollary of the Lemma 1. Indeed for any sequence σ in A^*, and for any formula φ we have $\sigma_{\downarrow A_\varphi} = (\sigma_{\downarrow A_\varphi})_{\downarrow A_\varphi}$.

A.3 Proof of the Theorem 1

The proof is done by structural induction on φ.

A.3.1 For the predicates

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

A.3.2 For the $\neg$ 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,

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

Since controller $\mathtt{C}_\neg$ 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:

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

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 $\mathtt{C}_\neg$'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.

A.3.3 For the $\wedge$ operator.

The principle for this proof is exactly the same, the general expression of the execution sequence is just a little bit more complex.

A.3.4 For the $\mathcal{U}$ operator.

Let suppose $\varphi = \varphi_l \mathcal{U} \varphi_r$. We have to prove that:

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

Let $\sigma \in \text{Exec}(GT(\varphi_l \mathcal{U} \varphi_r, \mathcal{L}) \otimes_{A_\varphi} I)$ suppose that $\text{VExec}(\sigma) = pass$.
For this controller we can not give a general expression of an execution sequence. We proceed by a backward analysis of the controller. In each state in which a *pass* verdict is emitted we follow the path leading to verdict emissions from the sub-controlled test in a way that the semantics of the tested formula is respected. As so, for each emission of verdict we give prominence to an execution sequence pattern that respects the semantics of our logic. First notice that the two instantiations of the $\mathtt{C}_x$ controller are doing the same job, they abstract the reception of verdict for the $\mathtt{C}_m$ controller and loop the sub-controlled tests which allows us to reason only on the $\mathtt{C}_m$.
Suppose that the $\mathtt{C}_\mathcal{U}$ emits a *pass* verdict via this state and consider $\sigma \in \text{Exec}(GT(\varphi_l \mathcal{U} \varphi_r, \mathcal{L}) \otimes_{A_\varphi} I)$, an execution sequence. We have to show for each possible emission of a *pass* verdict that the corresponding execution sequence satisfies the formula φ.

- Let us proceed first with the case shown on Fig. 13. A backward execution analysis underlines the red path on the $\mathtt{C}_m$ controller. An execution trace on this level (that is projected on the controller $controller_m$ must be in the form $c\_start() \cdot (l\_pass())^* \cdot r\_pass()$, which corresponds to an execution sequence $c\_start() \cdot (c\_start_l() \cdot \delta \cdot c\_ver_l(pass) \cdot l\_pass())^* \cdot (c\_start_r() \cdot \delta \cdot c\_ver_r(pass) \cdot r\_pass())$, where δ denotes some intermediate actions.. Applying the induction hypothesis on the subsequences $c\_start_l() \cdot \delta \cdot c\_ver_l(pass) \cdot l\_pass()$ and the subsequence $c\_start_r() \cdot \delta \cdot c\_ver_r(pass) \cdot r\_pass()$, that means for the global execution sequence σ that there exists $n \in \mathbb{N}$ s.t. $\sigma = \sigma_1 \cdots \sigma_n \cdot \sigma' \cdot \sigma''$ with $\forall i \in \{1 \cdots n\} \cdot \sigma_i \models \varphi_l$ and $\sigma' \models \varphi_r$. That is $\sigma \models \varphi$.

- Now we deal with the emission of a *pass* verdict throughout the state shown on Fig. 14. Starting from the Pass state and getting back on the automaton to the initial state we show a pattern that any execution sequence should have:

$$c\_start() \cdot (l\_pass())^* \cdot l\_fail() \cdot \big(l\_pass() \mid l\_fail()\big)^* \cdot ?r\_pass()$$

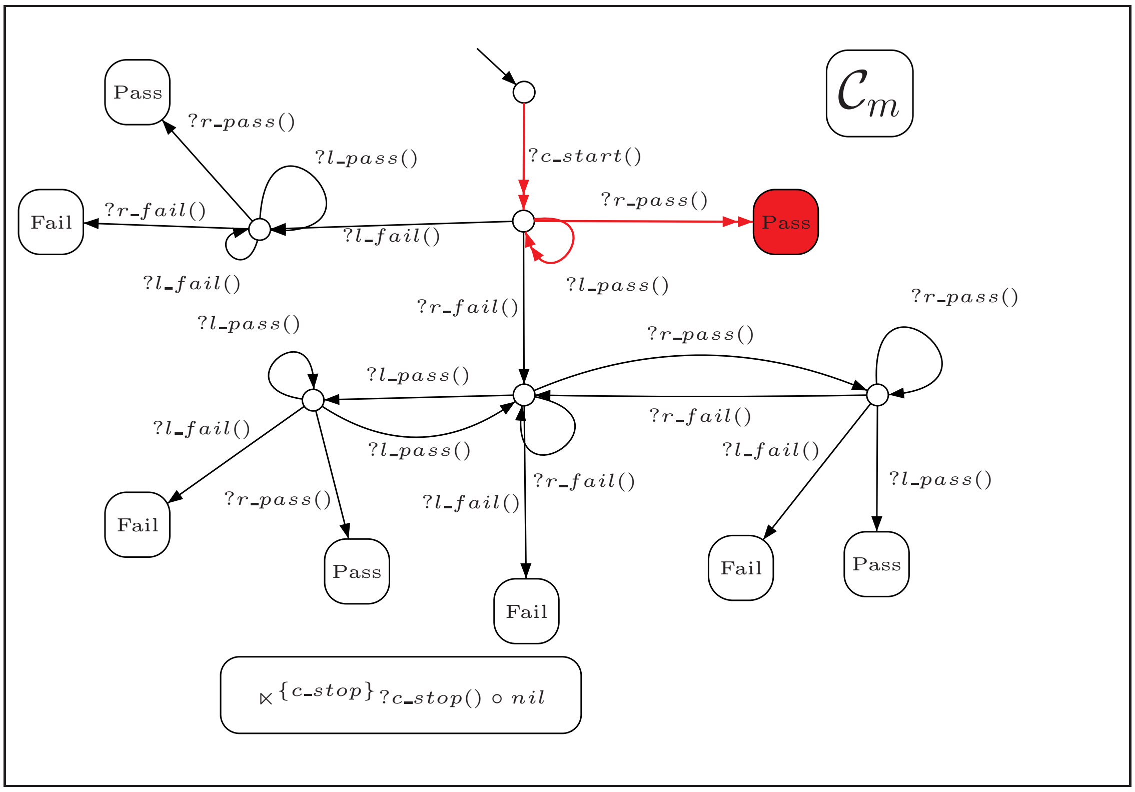

Figure 13: A path in the $\mathtt{C}_m$ part of the $\mathtt{C}_{\mathcal{U}}$ controller

That is without projection (omitting parenthesis for actions without arguments):

$$c\_start \cdot (c\_start_l \cdot \delta \cdot c\_ver_l(pass) \cdot l\_pass)^* \cdot (c\_start_l \cdot \delta \cdot c\_ver_l(fail) \cdot l\_fail) \cdot \pi^* \cdot (c\_start_r \cdot \delta \cdot c\_ver_r(pass) \cdot r\_pass)$$

with:

- $\pi = ((c\_start_l \cdot \delta \cdot c\_ver_l(pass) \cdot l\_pass) \mid (c\_start_l \cdot \delta \cdot c\_ver_l(fail) \cdot l\_fail))$
- δ denoting some intermediate actions.

Applying the induction hypothesis, that means σ is in the form $\sigma' \cdot \sigma''$, and it exists $n, m \in \mathbb{N}$ s.t.

$$\begin{array}{ll} & \sigma = \omega_0 \cdots \omega_n \cdot \omega_{\text{l\_}fail} \cdot \omega'_0 \cdots \omega'_m \\ \text{and} & \forall i \leq n \cdot \omega_{i\downarrow_{A_{\varphi_l}}} \models_T \varphi_l \\ \text{and} & \forall i \leq m \cdot (\omega'_{i\downarrow_{A_{\varphi_l}}} \models_T \varphi_l \vee \omega'_{i\downarrow_{A_{\varphi_l}}} \models_F \varphi_l \\ \text{and} & \left(\omega_0 \cdots \omega_n \cdot \omega_{\text{l\_}fail} \cdot \omega'_0 \cdots \omega'_m\right)_{\downarrow_{A_{\varphi_l}}} \models_T \varphi_r \end{array}$$

That means that σ can be view as $s \cdot s'$ with $s \models \varphi_r$. That is $\sigma \models_T \varphi_r$

- Now we treat the of an emission of a *pass* verdict via the state shown on Fig. 15. Using the same principle, one could see that the projection on $\mathtt{C}_m$ of the execution sequence σ must be in the form:

$$c\_start() \cdot (l\_pass())^* \cdot r\_fail() \cdot \left(l\_pass()^+ \cdot r\_fail() \mid r\_fail() \mid r\_pass()^+ \cdot r\_fail()\right)^* \cdot r\_pass()^+ \cdot l\_pass()$$

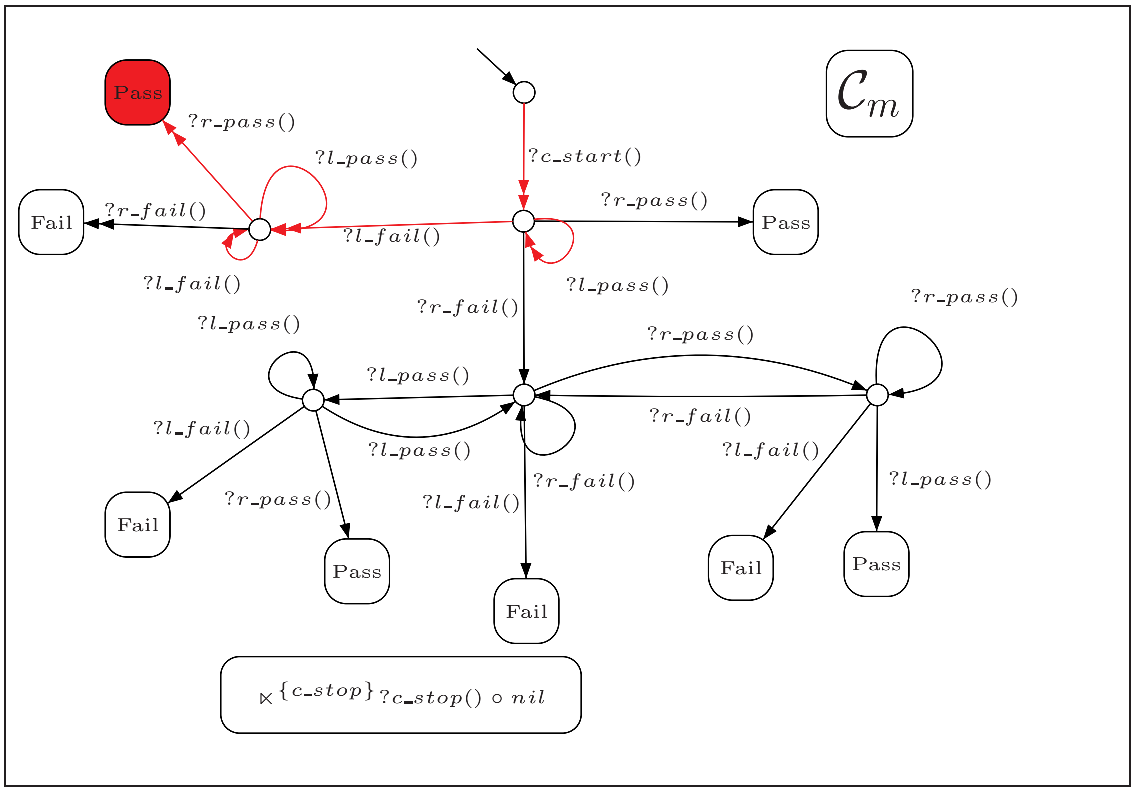

Figure 14: A path in the $\mathtt{C}_m$ part of the $\mathtt{C}_{\mathcal{U}}$ controller

That is without projection (omitting parenthesis for actions without arguments):

$$c\_start \cdot (c\_start_l \cdot \delta \cdot c\_ver_l(pass) \cdot l\_pass)^* \cdot (c\_start_r \cdot \delta \cdot c\_ver_r(fail) \cdot r\_fail) \cdot \pi^* \cdot r\_pass^+ \cdot l\_pass$$

with:

- $\pi = (c\_start_l \cdot \delta \cdot c\_ver_l(pass) \cdot l\_pass)^+ \cdot (c\_start_r \cdot \delta \cdot c\_ver_r(fail) \cdot r\_fail) \mid (c\_start_r \cdot \delta \cdot c\_ver_r(fail) \cdot r\_fail) \mid (c\_start_r \cdot \delta \cdot c\_ver_r(pass) \cdot r\_pass)^+ \cdot (c\_start_r \cdot \delta \cdot c\_ver_r(fail) \cdot r\_fail)$
- δ denoting some intermediate actions.

What is important to notice is that the last two verdicts received from the sub-test are a *pass* verdict from the left sub-formula ($?l\_pass()$) and previously *pass* verdicts for the right sub-formula.($?r\_pass()$). We can apply the induction hypothesis on each subsequence of σ producing a verdict for each sub-formula, we found then that the execution sequence is then in the form:

$$\begin{array}{ll} & \omega = \omega' \cdot \omega^{\heartsuit} \cdot \omega^{\clubsuit} \cdot \omega^{\spadesuit} \\ \text{with} & \omega^{\clubsuit} \models \varphi_r \\ \text{with} & \omega^{\heartsuit} \cdot \omega^{\clubsuit} \cdot \omega^{\spadesuit} \models \varphi_l \end{array}$$

Analyzing paths in ω', one can see that it does not contain $l\_fail()$. As so the last $l\_pass()$ received is the unique one, or there might be others in the trace. This can be easily shown by recurrence on the number of occurrence of the π pattern in the trace ω. That means that $\omega' \models \varphi_l$, and so as $\omega' \cdot \omega^{\heartsuit} \cdot \omega^{\clubsuit} \cdot \omega^{\spadesuit}$. That is $\omega \models \varphi$.

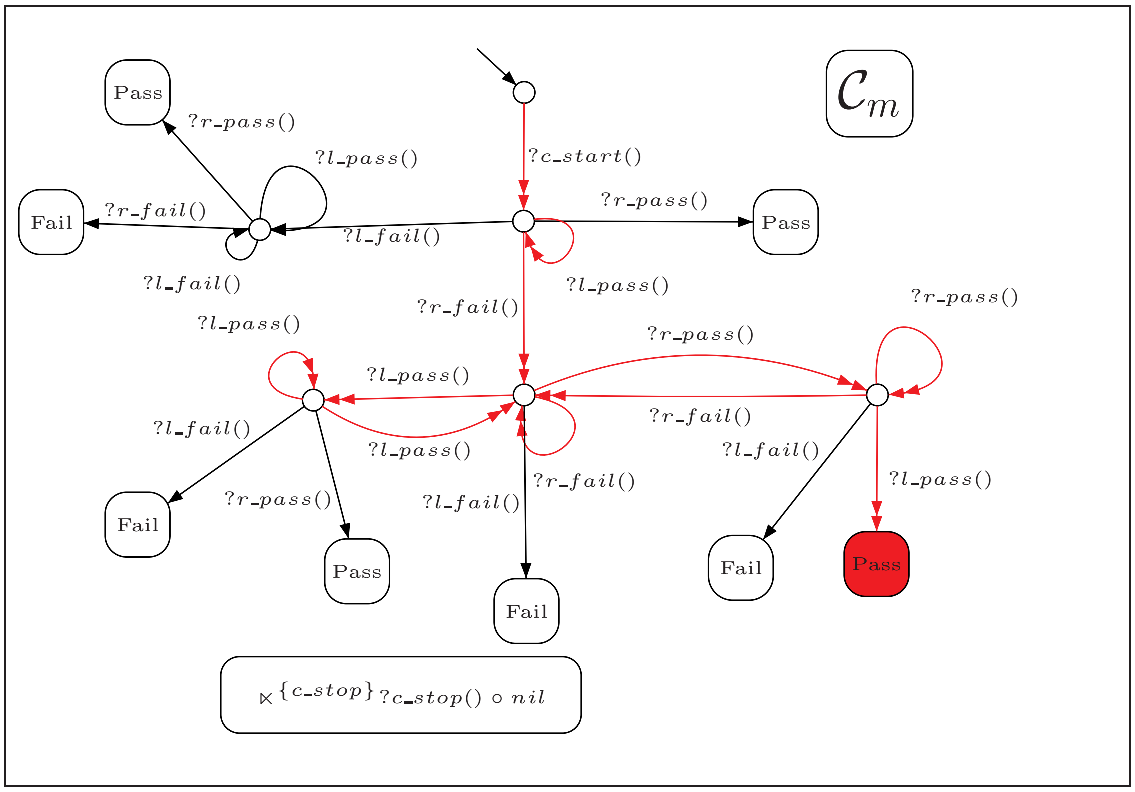

Figure 15: A path in the $\mathtt{C}_m$ part of the $\mathtt{C}_{\mathcal{U}}$ controller

- For the case shown on Fig. 16, the principle is exactly the same as the previous one.

B Proofs for the ERE instantiations

This section sketches proofs for the propositions given in the ERE instantiation of the framework. These proofs are naturally following the same way as the ones for the LTL instantiation. Each Lemma and the propostion are shown by a structural induction on the extended regular expressions. Basic cases of the induction are exactly the same as the previously presented ones as they scope the basic predicates. The proof step of a negation does not change either. For the choice operator $(+)$ it is also very similar as it corresponds to a disjunction. Indeed, either one could see the proof as a direct consequence of a combination using conjunction and negation used in the LTL's proofs. Or one could see that controllers only differ by the labelling edges. Proofs of the concatenation operator $(\cdot)$ and the Kleene star rely on a slicing of the sequences.

References

[1] Hartman, A.: Model based test generation tools survey. Technical report, AGEDIS Consortium (2002) 1

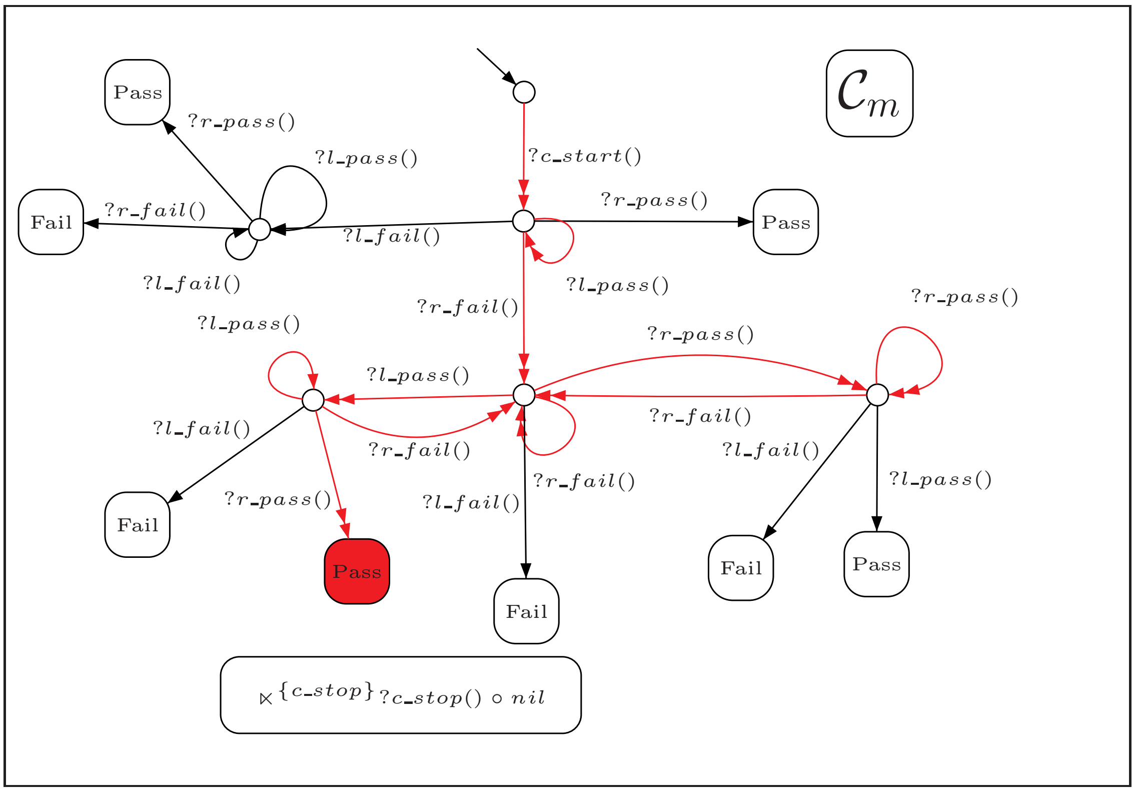

Figure 16: A path in the $\mathtt{C}_m$ part of the $\mathtt{C}_{\mathcal{U}}$ controller

[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). Volume 3205 of Lecture Notes in Computer Science., Springer-Verlag (2005) 81–96 1

[3] Darmaillacq, V., Fernandez, J.C., Groz, R., Mounier, L., Richier, J.L.: Test generation for network security rules. In: TestCom. (2006) 341–356 1

[4] Falcone, Y., Fernandez, J.C., Mounier, L., Richier, J.L.: A test calculus framework applied to network security policies. In: FATES/RV. (2006) 55–69 1, 2.3

[5] Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, London, UK, Springer-Verlag (2002) 342–356 1

[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). Volume 144(4) of ENTCS. (2005) 3–20 1, 4

[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** (2005) 209–234 1

[8] Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer-Verlag New York, Inc., New York, NY, USA (1995) 3, 3.1

[9] 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 3, 3.2

[10] Clarke, E., Grumberg, O., Peled, S.: Model Checking. The MIT Press (1997) 3.1