A Test Calculus Framework applied to network security policies

Yliès Falcone[1], Jean-Claude Fernandez[1], Laurent Mounier[1], and Jean-Luc Richier[2]
Email: {Ylies.Falcone,Jean-Claude.Fernandez,Laurent.Mounier, Jean-Luc.Richier}@imag.fr

[1] Vérimag Laboratory, Gières, France
[2] LSR-IMAG Laboratory, St Martin d'Hères, France

Abstract. We propose a syntax-driven test generation technique to automaticaly derive abstract test cases from a set of requirements expressed in a linear temporal logic. Assuming that an elementary test case (called a "tile") is associated to each basic predicate of the formula, we show how to generate a set of test controlers associated to each logical operator, and able to coordinate the whole test execution. The test cases produced are expressed in a process algebraic style, allowing to take into account the test environment constraints. We illustrate this approach in the context of network security testing, for which more classical model-based techniques are not always suitable.

1 Introduction

Testing is a very popular validation technique, used in various application domains, and for which several formalizations have been proposed. In particular, a well-defined theory is the one commonly used in the telecommunication area for *conformance testing* of communication protocols [1]. This approach, sometimes called "model-based" approach, consists in defining a conformance relation [2, 3] between a specification of the system under test and a model of its actual implementation. The purpose of the test is then to decide if this relation holds or not. A practical interest is that test cases can be automatically produced from this specification. Several tools implement this automatic generation technique, e.g. [4–7].

However, this model-based approach requires a rather complete specification of the system under test, defined on a precise interface level. If this requirement can be fulfilled for specific pieces of software (e.g., a communication protocol), it may be difficult to achieve for large systems, or when the system requirements cannot be encoded as a conformance relation defined on a single interface level. A typical example of such situation is testing a network security policy, where expected properties may rely on the whole network behavior (not on a single component or protocol), and can be tested only by accessing several interface levels.

In a previous work [8], we have proposed an alternative approach when the system requirements are expressed as a set of (temporal) logic formulae. For each formula ϕ, an abstract test case t_ϕ is produced following a syntax-driven technique: assuming that an elementary test case t_i (called hereafter a "tile") has been associated to each literal p_i of formula ϕ, the whole test case t_i is obtained by combining the tiles using test operators corresponding to the logical operators appearing in formula ϕ. This provides a structural correspondence between formulae and tests and it is easy to prove that the test obtained are sound with respect to the semantics of the formulae (in other words we give a "test based" semantics of the logic which is compatible with the initial one). The originality of this approach is then that a part of the system specification is encoded into the tiles, that can be provided by the system designer, or a by a test expert, which is assumed to be easier to obtain that a global specification. We illustrated this approach in the context of security testing.

This paper extends this previous work from the test execution point of view. In [8], abstract test cases were directly expressed by labelled transition systems, independently of the test architecture. We propose here to better take into account the test execution and to express the test cases in a higher level formalism. In particular we show how to produce well structured test cases consisting of a set of test drivers (one test driver for each elementary tile), coordinated by a set of test controllers (corresponding to the logical operators appearing in the formula). Thus, independent parts of the formula can be tested in parallel (either to speed up the test execution, or due to test environment constraints), each local verdicts being combined in a consistent way by the test controllers. Formally, test cases are expressed in a classical process algebra (called a "test calculus"), using basic control operators (parallel composition and interruption) and data types to handle test parameters and verdicts.

This paper is organized as follows: section 2 introduces our "test calculus" process algebra, and section 3 defines the notions of test execution and test verdicts. We propose in section 4 a simple temporal logic allowing to express network security requirements, and we show how to produce test cases from this logic in section 5. Finally, section 6 provides some examples in the context of network security policies.

2 Test process algebra

To model processes, we define a rather classic term algebra with typed variables, inspired from CCS [9], CSP [10] and Lotos. We suppose a set of predefined actions *Act*, a set of types $\mathcal{T}$, and a set of variables *Var*. Actions are either modifications of variables or communications through channels which are also typed. In the following, we do not address the problem of verifying that communications and assignments are well-typed. We denote by $expr_\tau$ (resp. x_τ) any expression (resp. variable) of type τ. Thus, when we write $x_\tau := expr_\tau$, we consider that this assignment is well typed.

A test is described as a term of our process algebra. We distinguish between elementary test cases, which are elements of a basic process algebra and compound test cases. We give the syntax and an operational semantics of this test process algebra.

2.1 Basic processes

Our basic process algebra allows to describe sequences of atomic actions, communication and iteration. A term of this algebra is called a tile, which are the elementary test components and we note *TILE* the set of all tiles.

The syntax of tiles and actions is given by the following grammar:

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

where $e \in TILE$ is a tile, b a boolean expression, c a channel name, γ an action, $\circ$ is the *prefixing* operator ($\circ : Act \times TILE \to TILE$), $+$ the *choice* operator, and $recX : TILE \to TILE$ allows *recursive* tile definition with X a term variable. When the condition b is true, we abbreviate $[true]\gamma$ by γ. The special tile nil does nothing.

There are two kinds of actions ($\gamma \in Act$). The first ones are the internal actions (modification of variables). The second ones are the communication actions. Two kinds of communications exist: $?c(x_\tau)$ denotes value reception on a channel c which is stored in variable x_τ; $!c(expr_\tau)$ denotes the emission of a value $expr_\tau$ on a channel c. Communication is done by “rendez-vous”.

2.2 Composing processes

Processes are compositions of tiles. Choices we made about composition operators came from needs appearing in our case studies in network security policies [8]. Composing tests in sequence is quite natural; however, for independent actions, and in order to speed-up test executions, one might want to parallelize some tests executions, for example, if one wants to scan several computers on a network. The parallel composition is also used to model the execution and communication between the test processes and the rest of the system. We assume a set $\mathcal{C}$ of channels used by tiles to communicate. We distinguish internal channels (set $\mathcal{C}_{\text{in}}$) and external channels (set $\mathcal{C}_{\text{out}}$), and we have $\mathcal{C} = \mathcal{C}_{\text{in}} \cup \mathcal{C}_{\text{out}}$.

In case of several processes executing in parallel, one might want to interrupt them. We choose to add an operator providing an exception mechanism: it permits to replace a process by an other one on the reception of a communication signal.

So, we define a set of operators, $\{\|_{\mathcal{L}}, \ltimes^{\mathcal{I}}\}$, respectively the parallel (with communication through a channel list $\mathcal{L} \subseteq \mathcal{C}$), and exception (with an action list $\mathcal{I}$) compositions.

The grammar for term processes (*TERM*) is:

$$t ::= e \mid t \parallel_{\mathcal{L}} t \mid t \ltimes^{\mathcal{I}} t$$

The parallel operator $\parallel_{\mathcal{L}}$ (so as the choice operator $+$) is associative and commutative. It expresses either the interleaving of independant action or the emission $!c(expr_{\mathcal{T}})$ of the value of an expression $expr_{\mathcal{T}}$ on a channel c. When the value is received by a process $?c(x_{\mathcal{T}})$, the communication is denoted at the syntactic level by $c(expr_{\mathcal{T}}/x_{\mathcal{T}})$. The independent and parallel execution $\parallel_{\emptyset}$ is noted $\parallel$.

The Join-Exception operator $\ltimes^{\mathcal{I}}$ is used to interrupt a process and replace it with an other when a synchronization/global/communication action belonging to its synchronization list $\mathcal{I}$ occurs. Intuitively, considering two processes t, t' and a communication action α, $t \ltimes^{\{\alpha\}} t'$ means that if α is possible, t is replaced by t', else t continues normally.

2.3 Semantics

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

$$\frac{\alpha \in Act \qquad t_2 \xrightarrow{\alpha} t_2'}{t_1 + t_2 \xrightarrow{\alpha} t_2'}\ (+)$$

$$\frac{\alpha \notin \{[b]!c(expr_{\mathcal{T}}), [b]?c(x_{\mathcal{T}}) | c \in \mathcal{L}, b \in \mathbb{B}_{exp}\} \qquad t_1 \xrightarrow{\alpha} t_1'}{t_1 \parallel_{\mathcal{L}} t_2 \xrightarrow{\alpha} t_1' \parallel_{\mathcal{L}} t_2}\ (\parallel_{\neg\mathcal{L}})$$

$$\frac{c \in \mathcal{C}_{in} \wedge c \in \mathcal{L} \qquad t_1 \xrightarrow{[b]!c(expr_{\mathcal{T}})} t_1' \qquad t_2 \xrightarrow{[b]?c(x_{\mathcal{T}})} t_2'}{t_1 \parallel_{\mathcal{L}} t_2 \xrightarrow{[b]c(expr_{\mathcal{T}}/x_{\mathcal{T}})} t_1' \parallel_{\mathcal{L}} t_2'}\ (\parallel_{\mathcal{C}_{in}})$$

$$\frac{c \in \mathcal{C}_{out} \wedge c \in \mathcal{L} \qquad t_1 \xrightarrow{[b]!c(expr_{\mathcal{T}})} t_1'}{t_1 \parallel_{\mathcal{L}} t_2 \xrightarrow{[b]!c(expr_{\mathcal{T}})} t_1' \parallel_{\mathcal{L}} t_2}\ (!\ \parallel_{\mathcal{C}_{out}})$$

$$\frac{c \in \mathcal{C}_{out} \wedge c \in \mathcal{L} \qquad t_1 \xrightarrow{[b]?c(x_{\mathcal{T}})} t_1'}{t_1 \parallel_{\mathcal{L}} t_2 \xrightarrow{[b]?c(expr_{\mathcal{T}}/x_{\mathcal{T}})} t_1' \parallel_{\mathcal{L}} t_2}\ (?\ \parallel_{\mathcal{C}_{out}})$$

$$\frac{\alpha \in \mathcal{I} \qquad t_2 \xrightarrow{\alpha} t_2'}{t_1 \ltimes^{\mathcal{I}} t_2 \xrightarrow{\alpha} t_2'}\ (\ltimes^{\mathcal{I}}_{\alpha}) \qquad \frac{\alpha \notin \mathcal{I} \qquad t_1 \xrightarrow{\alpha} t_1'}{t_1 \ltimes^{\mathcal{I}} t_2 \xrightarrow{\alpha} t_1' \ltimes^{\mathcal{I}} t_2}\ (\ltimes^{\mathcal{I}}_{\neg\alpha})$$

Fig. 1. Rules for term rewriting

A runtime environment ρ maps the set of variables to the set of values. We note $\mathcal{E}$ the set of all environments. Actions modify environments in a classical way; we note $\rho \xrightarrow{\gamma} \rho'$ the modification of environment ρ into ρ' by action γ. For example, $\rho \xrightarrow{x_{\mathcal{T}}:=exp_{\mathcal{T}}} \rho[\rho(exp_{\mathcal{T}})/x_{\mathcal{T}}]$, where $\rho[\rho(exp_{\mathcal{T}})/x_{\mathcal{T}}]$ is the environment ρ in which variable $x_{\mathcal{T}}$ is associated the value $\rho(exp_{\mathcal{T}})$. In the following, environments are extended to any typed expression.

A *labelled transition system* (LTS, for short) is a quadruplet (Q, A, T, q^0) where Q is a set of states, A a set of labels, T the transition relation ($T \subseteq Q \times A \times Q$) and q^0 the initial state ($q^0 \in Q$). We will use the following definitions and notations: $(p, a, q) \in T$ is noted $p \xrightarrow{a}_T q$ (or simply $p \xrightarrow{a} q$). An *execution sequence* λ is a composition of transitions: $q^0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n$. We denote by σ^λ (resp. α^λ) the sequence of states (resp. observable actions) associated with λ. The sequence of actions α^λ is called a *trace*. We note by Σ_S, the set of finite execution sequences starting from the initial state q^0 of S. For any sequence λ of length n, λ_i or $\lambda(i)$ denotes the i-th element and $\lambda_{[i \cdots n]}$ denotes the suffix $\lambda_i \cdots \lambda_n$.

The semantics of a process is based on a LTS where states are "configurations", pairs (t, ρ), t being a term of the process algebra, ρ an environment, and transitions are given by definition 2. Configurations are used to represent process evolutions. We note $\mathcal{C}_{term} \overset{\text{def}}{=} TERM \times \mathcal{E}$ the set of configurations.

Definition 1 (Term-transition). *A term rewriting transition $\rightharpoonup$ is an element of $TERM \times Act \times TERM$. We say that the term t is rewritten in t' by action α. We note: $t \overset{\alpha}{\rightharpoonup} t'$. This semantics is similar with the CCS one [9].*

Term-transitions are defined in Figure 1 (using the fact that $\|$ is commutative).

Definition 2 (Transitions). *A transition is an element of $\mathcal{C}_{term} \times Act \times \mathcal{C}_{term}$. We say that the term t in the environment ρ is rewritten in t' modifying the environment ρ in ρ'.*

We have four transition rules, one for an assignment, and three for communication exchange. They are defined in Figure 2.

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

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

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

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

Fig. 2. Rules for environment modification

3 Test execution and test verdicts

As seen in the previous section, the semantics of a test case represented by a $TERM$ process t is expressed by a LTS $S_t = (Q^t, A^t, T^t, q^t_0)$. We assume here that the behaviour of the System Under Test (SUT) is also modelled by a LTS $I = (Q^I, A^I, T^I, q_0^I)$. A *test execution* is then a sequence of interactions between t and the SUT to deliver a *verdict* indicating whether the test succeeded or not. We first explain how verdicts are computed in our context, and then we give a formal definition of a test execution.

3.1 Tiles verdicts

We assume in the following that any elementary tile t_i owns at least one variable used to store its *local verdict*, namely a value of enumerated type $Verdict = \{pass, fail, inc\}$. This variable is supposed to be set to one of these values when tile execution terminates. The intuitive meaning we associate to each of these values is similar to the one used in conformance testing:

- *pass* means that the test execution of t_i did not reveal any violation of the requirement expressed by t_i;
- $fail$ means that the test execution of t_i did reveal a violation of the requirement expressed by t_i;
- *inc* means that the test execution of t_i did not allow to conclude about the validity of the requirement expressed by t_i.

We now have to address the issue of combing the different verdicts obtained by each tile execution of a whole test case.

3.2 Verdict management

The solution we adopt is to include in the test special processes (called *test controlers*) for managing tile verdicts. When tiles end their execution, i.e. have computed a verdict, they emit it toward a designated test controler which captures it. Depending on verdicts received, the controller emits a final verdict – and may halt the executions of some tests if they are not needed anymore. The "main" controler then owns a variable v_g to store the final verdict.

Test controllers can easily be written in our process algebra with communication operations as shown on the following example. The whole test case is then expressed as a term of our process algebra (with parallel composition and interuptions between processes).

An example of test controller Let us consider a test controller waiting to receive two pass verdicts in order to decide a global *pass* verdict (in other cases, it emits the last verdict received). Let c_v, be the channel on which verdicts are waited. The environment of this controller contains three variables, v for the verdicts received, v_g for the global verdict, and N to count numbers of verdicts remaining. An LTS representation is shown in Figure 3 and a corresponding algebraic expression is:

$$\mathtt{C} \overset{\text{def}}{=} recX$$
$$?c\_v_i(v_i) \circ ([v_i = pass]\ N\text{--} \circ X + [v_i \in \{inc, fail\}]\ v_g := v_i \circ\ !c\_v_g(v_g) \circ\ !Stop() \ \circ nil$$
$$+$$
$$[N = 0]\ v_g := pass \circ\ !c\_v_g(v_g) \circ\ !Stop() \ \circ nil$$

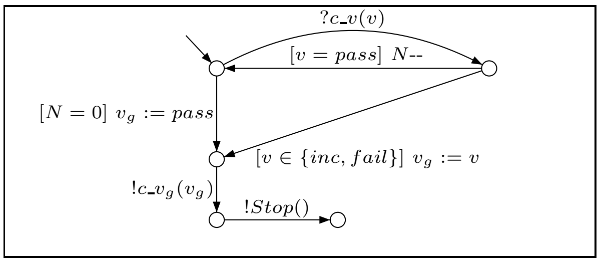


Fig. 3. Verdict controller combining pass verdicts.

3.3 Test execution

An execution of a test t (modelled by an LTS S_t) on a SUT (modelled by a LTS I), noted $\mathrm{Exec}(t, I)$, is simply expressed as a set of common execution sequences of S_t and I, defined by a composition operator $\otimes$.

Let $\lambda_I = q_0^I \xrightarrow{a_1} q_1^I \xrightarrow{a_2} q_2^I \cdots \xrightarrow{a_n} q_n^I \cdots \in \Sigma_I$ and $\lambda_{S_t} = q^{0,t} \xrightarrow{a_1} q_1^t \xrightarrow{a_2} q_2^t \cdots \xrightarrow{a_n} q_n^t \in \Sigma_{S_t}$, then $\lambda_{S_t} \otimes \lambda_I = (q^{0,t}, q_0^I) \xrightarrow{a_1} (q_1^t, q_1^I) \cdots \xrightarrow{a_n} (q_n^t, q_n^I) \in \mathrm{Exec}(t, I)$.

Let $\Sigma_{S_t}^{\text{pass}}$ (resp. $\Sigma_{S_t}^{\text{fail}}, \Sigma_{S_t}^{\text{inconc}}$) be the sets of states of S_t where variable v_g is set to *pass* (resp. $fail$, *inc*):

$$\begin{aligned}
\Sigma_{S_t}^{\text{pass}} &= \{(r, \rho) \mid \rho(v_g) = pass\} \\
\Sigma_{S_t}^{\text{fail}} &= \{(r, \rho) \mid \rho(v_g) = fail\} \\
\Sigma_{S_t}^{\text{inc}} &= \{(r, \rho) \mid \rho(v_g) = inc\}
\end{aligned}$$

For $\lambda \in \mathrm{Exec}(t, I)$, we define the verdict function: $\mathrm{VExec}(\lambda) = pass$ (resp. *fail*, *inconc*) iff there is $\lambda_{S_t} \in \Sigma_{S_t}^{\text{pass}}$ (resp. $\Sigma_{S_t}^{\text{fail}}, \Sigma_{S_t}^{\text{inconc}}$) and $\lambda_I \in \Sigma_I$ such that $\lambda_{S_t} \otimes \lambda_I = \lambda$.

4 Security rules formalization

In a previous work, we carried out a case study to analyse the network security policy in a university environment. This case study gave us a set of security requirements that could be expressed using a simple temporal logic. We give here the syntax and semantics of this logic.

4.1 Syntax

A security policy rule is expressed by a logical *formula* (φ), built upon *literals*. Each literal can be either a *condition literal* ($p_c \in P_c$), or an *event literal* ($p_e \in P_e$). A condition literal is a (static) predicate on the network configuration (e.g., *extRelay*(h) holds iff machine h is configured as an external relay), and an event literal corresponds to the occurrence of a transition in the network behavior (e.g.,

$enterNetwork(m)$ holds if message m is received by the network). A conjunction of condition literals is simply called a *condition* (C), whereas a conjunction of a single event literal and a condition is called a *(guarded) event* (E). The abstract syntax of a formula is given in Table 1. The intuitive meaning of these formulae is the following:

– An $\mathcal{O}$-Rule expresses a *conditional obligation*: when a particular condition holds, then another condition should also hold (logical implication).
– An $\mathcal{O}_T$-Rule expresses a *triggered obligation*: when a given event happens, then another condition should hold (or some event should occurs) before expiration of a given amount of time.
– An $\mathcal{F}$-Rule expresses an *interdiction*: when a given condition holds, or when a given event happens, then a given event is always prohibited.

$$\begin{array}{lll}
\varphi & ::= C \Rightarrow \mathcal{O}\, C & (\mathcal{O}\text{-Rule })\\
 & \mid\;\; E \Rightarrow \mathcal{O}_T\, C \mid E \Rightarrow \mathcal{O}_T\, E & (\mathcal{O}_T\text{-Rule})\\
 & \mid\;\; C \Rightarrow \mathcal{F}\, C \mid C \Rightarrow \mathcal{F}\, E & (\mathcal{F}\text{-Rule})\\
E & ::= p_e[C] \mid p_e & (\text{Event})\\
C & ::= \bigwedge_{i=1}^{n} p_{c_i} & (\text{Condition})
\end{array}$$

Table 1. Syntax of logic formulae

4.2 Semantics

Formulas are interpreted over LTS. Intuitively, a LTS S satisfies a formula φ iff *all* its execution sequences λ do, where condition literals are interpreted over *states*, event literals are interpreted over *labels*. We first introduce two interpretation functions for condition and event literals:
$f_c : P_c \rightarrow 2^Q$, associates to p_c the set of states on which p_c holds;
$f_e : P_e \rightarrow 2^A$, associates to p_e the set of labels on which p_e holds.

The satisfaction relation of a formula φ on an execution sequence λ ($\lambda \models \varphi$) is then (inductively) defined as follows:

– $\lambda \models C$ for $C = p_c^1 \wedge \cdots \wedge p_c^n$ iff $\forall i.\ \sigma^\lambda(1) \in f(p_c^i)$
– $\lambda \models p_e$ iff $\alpha^\lambda(1) \in g(p_e)$
– $\lambda \models p_e[C]$ iff $(\alpha^\lambda(1) \in g(p_e) \wedge \lambda(2) \models C)$
– $\lambda \models \varphi_1 \Rightarrow \mathcal{O}\, \varphi_2$ iff $((\lambda \models \varphi_1) \Rightarrow (\lambda \models \varphi_2))$
– $\lambda \models \varphi_1 \Rightarrow \mathcal{O}_T \varphi_2$ iff $((\lambda \models \varphi_1) \Rightarrow (\exists j \in [1, |\lambda|].\ \lambda(j) \models \varphi_2))$
– $\lambda \models \varphi_1 \Rightarrow \mathcal{F} \varphi_2$ iff $((\lambda \models \varphi_1) \Rightarrow (\forall j \in [1, |\lambda|].\ \lambda(j) \not\models \varphi_2))$

Finally, $S \models \varphi$ iff $\forall \lambda \in \Sigma_S.\ \lambda \models \varphi$.

5 Test generation

We define a structural generation function *GenTest* to convert a rule into the desired combination of elementary tiles with controllers. It associates controllers in such a way that the final verdict is *pass* iff the rule is satisfied by the SUT. Each controller emits its verdict on a channel, and may uses variables. In the following, new variables and channels will be silently created whenever necessary.

GenTest generates parallel and architecturally independent subtests. Formula semantics is ensured by the controller verdict combinations. Suitable scheduling of subtests is supplied by the controllers through channels used to start and stop subtests (given below by *Test* function).

5.1 Test generation function *GenTest*

Transformation of tiles Given a tile t_p (computing its verdict in the variable *ver*) associated to an elementary predicate p, the *Test* function transforms it. Intuitively, $Test(t_p, \mathcal{L})$, where $\mathcal{L}$ is a channel list, is t_p modified in order to be controlled through the channel list $\mathcal{L}$. More formally:

$$Test(t_p, \{c\_start, c\_stop, c\_loop, c\_ver\}) \overset{\text{def}}{=} recX\ (?c\_start() \circ t_p \circ (?c\_loop() \circ X + !c\_ver(ver) \circ nil)) \ltimes^{\{?c\_stop()\}} ?c\_stop() \circ nil$$

A representation on a LTS is shown in Figure 4.

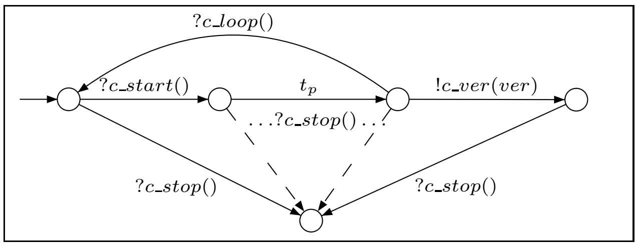


Fig. 4. Extension of tile t_p in a testing form

***GenTest* definition** The rule general form is: $P_l \Rightarrow \mathcal{M}\, P_r$ where $P_l, P_r \in \{E, C\}$ are predicates and $\mathcal{M} \in \{\mathcal{O}, \mathcal{O}_T, \mathcal{F}\}$ a modality.

The *GenTest* function is defined on the rule structure, giving an expression to be instantiated according to the different modalities. We suppose that the final verdict is emitted on the *main* channel, and t_{c_i}, t_{p_e} are the tiles respectively associated to elementary predicates c_i, p_e.

$$GenTest(P_l \Rightarrow \mathcal{M}\, P_r) \overset{def}{=} (GenTest_P(P_l, \mathcal{L}_l) \parallel GenTest_P(P_r, \mathcal{L}_r)) \parallel_{\mathcal{L}} \complement_{\mathcal{M}}(\mathcal{L}_l, \mathcal{L}_r)$$
$$\text{with } \mathcal{L} = \mathcal{L}_l \cup \mathcal{L}_r,$$
$$\mathcal{L}_l = \{c\_start_l, c\_stop_l, c\_loop_l, c\_ver_l\},$$
$$\mathcal{L}_r = \{c\_start_r, c\_stop_r, c\_loop_r, c\_ver_r\}$$

$$
\begin{aligned}
&GenTest_P(E, \{c\_start, c\_stop, c\_loop, c\_ver\}) \overset{def}{=} \\
&\quad \text{if } E = p_e[C], \quad \big(Test(t_{p_e}, \mathcal{L}_e) \parallel GenTest_C(C, \mathcal{L}_c)\big) \\
&\qquad\qquad\qquad\qquad \parallel_{\mathcal{L}} \mathsf{C}_E(\{c\_start, c\_stop, c\_loop, c\_ver\}, \mathcal{L}_e, \mathcal{L}_c) \\
&\qquad\qquad\qquad\qquad \text{with } \mathcal{L} \overset{\text{def}}{=} \mathcal{L}_e \cup \mathcal{L}_c \\
&\qquad\qquad\qquad\qquad\qquad \mathcal{L}_e = \{c\_start_e, c\_stop_e, c\_loop_e, c\_ver_e\}, \\
&\qquad\qquad\qquad\qquad\qquad \mathcal{L}_c = \{c\_start_c, c\_stop_c, c\_loop_c, c\_ver_c\} \\
&\quad \text{else /* } E = p_e \text{ */} \quad Test(t_{p_e}, \{c\_start, c\_stop, c\_loop, c\_ver\})
\end{aligned}
$$

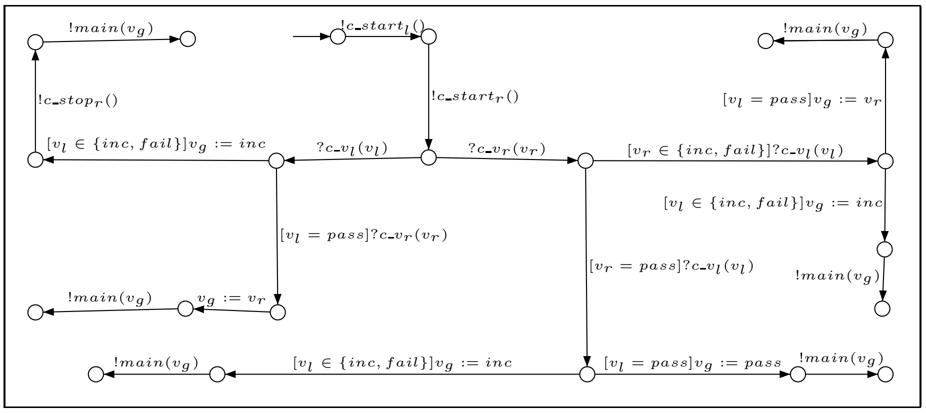
$$
GenTest_P(C, \mathcal{L}) \overset{\text{def}}{=} GenTest_C(C, \mathcal{L})
$$

$$
\begin{aligned}
&GenTest_C(\wedge_{i=1}^{n} c_i, \{c\_start, c\_stop, c\_loop, c\_ver\}) \overset{def}{=} \\
&\quad \text{if } n = 1, \quad Test(t_{c_1}, \{c\_start, c\_stop, c\_loop, c\_ver\}) \\
&\quad \text{else /* } n > 1 \text{ */} \\
&\qquad\quad \parallel_{i=1}^{n} Test(t_{c_i}, \mathcal{L}_i) \parallel_{\mathcal{L}} \mathsf{C}_\wedge(\{c\_start, c\_stop, c\_loop, c\_ver\}, (\mathcal{L}_i)_{i=1\ldots n}, n) \\
&\qquad\quad \text{with } \mathcal{L} = \cup_{i=1}^{n} \mathcal{L}_i; \forall i \in \{1 \ldots n\}, \mathcal{L}_i = \{c\_start, c\_stop, c\_loop, c\_ver_i\}
\end{aligned}
$$

5.2 Verdict controllers

Several verdict controllers are used in the *GenTest* definition. Controllers have different purposes. They are first used to manage the execution of subtests corresponding to the components of the rule. For example, for a $\mathcal{O}_T$ formula, we have to wait for the left-side subtest before starting the right-side subtest. Controllers are also used to "implement" the formula semantics by combining verdicts from subtests.

Controllers definitions are parameterized with channel parameters. We give here an informal description of the controllers, with a graph definition for the more important ones; other controllers are similar (definitions given in appendix).



$!main(v_g)$
$!c\_start_l()$
$!main(v_g)$
$!c\_stop_r()$
$!c\_start_r()$
$[v_l = pass]v_g := v_r$
$[v_l \in \{inc, fail\}]v_g := inc$
$?c\_v_l(v_l)$
$?c\_v_r(v_r)$
$[v_r \in \{inc, fail\}]?c\_v_l(v_l)$
$[v_l \in \{inc, fail\}]v_g := inc$
$[v_l = pass]?c\_v_r(v_r)$
$[v_r = pass]?c\_v_l(v_l)$
$!main(v_g)$
$!main(v_g)$
$v_g := v_r$
$!main(v_g)$
$[v_l \in \{inc, fail\}]v_g := inc$
$[v_l = pass]v_g := pass$
$!main(v_g)$

Fig. 5. An instantiated LTS representation of the $\mathsf{C}_{\mathcal{O}}$ controller.

Formula level controllers They emit their verdict on the channel *main*.

1. $\mathtt{C}_{\mathcal{O}}(channel\_list, channel\_list)$. This controller is used to manage the execution of tiles corresponding to the left and right part of a static implication. The controller starts the two tests corresponding to the two sides of the implication. Then it waits for the reception of a verdict (verdicts can arrive in any order). According to the semantics of implication and the first verdict received, it decides either to wait for the second verdict or to emit a verdict immediately. The controller takes two channel lists as parameters for managing the execution and verdict of each side of the implication. The associated environment contains three variables. A LTS representation of $\mathtt{C}_{\mathcal{O}}(\{c\_start_l, c\_stop_l, c\_v_l, c\_loop_l\}, \{c\_start_r, c\_stop_r, c\_v_r, c\_loop_r\})$ is shown in Figure 5.
2. $\mathtt{C}_{\mathcal{O}_T}(channel\_list, channel\_list)$. This controller is used to manage the execution of tiles corresponding to the sides of an implication with a triggered obligation. The controller starts the test corresponding to the left side of the implication. If this test is inconclusive or fails, a *inc* verdict is decided. Otherwise, the timer and the second test are started. If the test emits *pass*, the final verdict is *pass*. As long as the timer is not expired, (that is, the boolean variable *t_out* is false), if the second test ends with *fail* or *inc*, the test is started again. When the the timer expires, a stop signal ($!c\_stop_r$) is sent to the right side test. In that case, the final verdict is *inc* if an *inc* verdict occured, *fail* otherwise. A LTS representation of $\mathtt{C}_{\mathcal{O}_T}(\{c\_start_l, c\_stop_l, c\_loop_l, c\_ver_l\}, \{c\_start_r, c\_stop_r, c\_loop_r, c\_ver_r\})$ is shown in Figure 6.

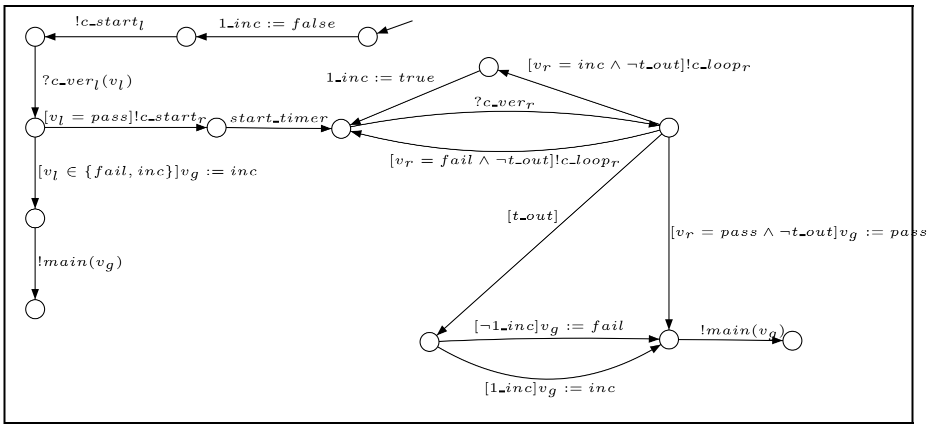

$!c\_start_l$
$1\_inc := false$
$?c\_ver_l(v_l)$
$1\_inc := true$
$[v_r = inc \wedge \neg t\_out]!c\_loop_r$
$?c\_ver_r$
$[v_l = pass]!c\_start_r$
$start\_timer$
$[v_r = fail \wedge \neg t\_out]!c\_loop_r$
$[v_l \in \{fail, inc\}]v_g := inc$
$[t\_out]$
$[v_r = pass \wedge \neg t\_out]v_g := pass$
$!main(v_g)$
$[\neg 1\_inc]v_g := fail$
$!main(v_g)$
$[1\_inc]v_g := inc$

Fig. 6. An instantiated LTS representation of the $\mathtt{C}_{\mathcal{O}_T}$ controller

3. $\mathtt{C}_{\mathcal{F}}(channel\_list, channel\_list)$. This controller is similar to the $\mathtt{C}_{\mathcal{O}}$ controller. It waits for a fail verdict for the right-side subtest in order to conclude on a *pass* verdict. A LTS definition of this controller is given in appendix A.2 in Figure 7.

Predicate level controllers

1. $\mathtt{C}_E(channel\_list, channel\_list, channel\_list)$. This controller is used to manage executions and verdicts around an event. The controller starts the execution of the event, and then, depending on the verdict received, it starts the subtests associated to the condition predicates in E. This conditions have to be tested after the event. A LTS definition of this controller is given in appendix A.3 in Figure 8.
2. $\mathtt{C}_\wedge(channel\_lists, integer)$. Informally this controller starts different test and waits for verdicts. Like the other controllers it controls subtests with a channel. If all tests succeed, the controller emits a *pass* verdict. If some tests do not respond *pass* the controller emits the last verdict received and stops the other potentially executing subtests. This controller is a generalization of the one presented in 3.1. A LTS definition of this controller is given in appendix A.4 in Figure 9.

5.3 Soundness proposition

We now express that an abstract test case produced by function $GenTest$ is always *sound*, i.e. it delivers a *fail* verdict when executed on a network behavior I only if formula ϕ does not hold on I. To do this, we follow a very similar approach than in [8]. Two hypotheses are required in order to prove this soundness property:

H1. First, for any formula φ, we assume that each elementary test case t_i provided for the (event or condition) literals p_i appearing in φ is *strongly sound* in the following sense:
Execution of t_i on SUT I always terminate, and
$\forall \lambda \in \text{Exec}(t_i, I) \cdot \text{VExec}(\lambda) = \mathit{Pass} \Rightarrow \lambda \models p_i \wedge (\text{VExec}(\lambda) = \mathit{Fail} \Rightarrow \lambda \not\models p_i)$.

H2. Second, we assume that the whole execution of a (provided or generated) test case t associated to a condition C is *stable* with respect to condition literals: the valuation of these literal does not change during the test execution. This simply means that the network configuration is supposed to remain stable when a condition is tested. Formally:
$\forall p_i \in P_c. \forall \lambda \in \Sigma_I \cdot \lambda_{S_t} \otimes \lambda \in \text{Exec}(t, I) \Rightarrow (\sigma^\lambda \subseteq f_c(p_i) \vee \sigma^\lambda \cap f_c(p_i) = \emptyset)$
where σ^λ denotes here tacitly a set of states instead of a sequence.

We now formulate the soundness property:
Proposition: Let φ a formula, I an LTS and $t = \text{GenTest}(\varphi)$. Then:
$\lambda \in \text{Exec}(t, I) \wedge \text{VExec}(\lambda) = fail \Longrightarrow I \not\models \varphi$.

6 Application

In this section we apply the $GenTest$ function with two rule patterns taken from the case study presented in [8].

6.1 $\mathcal{O}$-Rule

Consider the requirement *“External relays shall be in the DMZ”*, this could be reasonably understood as *“If a host is an external relay, it has to be in the DMZ”*. A possible modelisation is:

$$extRelay(h) \Rightarrow \mathcal{O}\,(inDMZ(h))$$

We suppose that somehow we have tiles models for $extRelay(h)$ and $inDMZ(h)$. The $GenTest$ function can be applied on the formula:

$$\begin{aligned}
&GenTest(extRelay(h) \Rightarrow \mathcal{O}\,(inDMZ(h)))\\
&\qquad = \Big(GenTest_P(extRelay(h), \mathcal{L}_l) \parallel GenTest_P(inDMZ(h), \mathcal{L}_r)\Big) \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}}(\mathcal{L}_l, \mathcal{L}_r)\\
&\qquad = \Big(GenTest_C(extRelay(h), \mathcal{L}_l) \parallel GenTest_C(inDMZ(h), \mathcal{L}_r)\Big) \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}}(\mathcal{L}_l, \mathcal{L}_r)\\
&\qquad = \Big(Test(extRelay(h), \mathcal{L}_l) \parallel Test(inDMZ(h), \mathcal{L}_r)\Big) \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}}(\mathcal{L}_l, \mathcal{L}_r)\\
&\quad \text{where } \ \mathcal{L} = \mathcal{L}_l \cup \mathcal{L}_r\\
&\qquad\quad \mathcal{L}_l = \{c\_start_l, c\_stop_l, c\_v_l, c\_loop_l\}\\
&\qquad\quad \mathcal{L}_r = \{c\_start_r, c\_stop_r, c\_v_r, c\_loop_r\}
\end{aligned}$$

The definitions of $t_{extRelay(h)}$ and $t_{inDMZ(h)}$ (writing their verdict in their verdict variable (ver)) in our algebra are:

$$\begin{aligned}
&t_{extRelay(h)} \overset{\text{def}}{=}\\
&\quad !connect(he, h) \circ\\
&\qquad \big(\,(?ok \circ !transfer(he, h, m) \circ [?ok \circ !tranfer(h, hi, m) \circ ver := pass + ?ko \circ ver := fail])\\
&\quad + (?ko \circ ver := fail)\\
&\quad \circ nil
\end{aligned}$$

$$\begin{aligned}
&t_{inDMZ(h)} \overset{\text{def}}{=}\\
&\quad !tracert(hd, h) \circ\\
&\qquad\quad ?end \circ ver := pass\\
&\qquad + \big[?resp(hx) \circ \big((?resp(hx') \circ ?end \circ ver := inc) + (?end \circ ver := fail)\big)\big]\\
&\quad \circ nil
\end{aligned}$$

$$\begin{aligned}
\text{So, }\ & GenTest(extRelay(h) \Rightarrow \mathcal{O}\ inDMZ(h)) =\\
&\qquad ((?c\_start_l \circ !connect(he, h) \circ \ldots) \ltimes^{\mathcal{I}_l} ?c\_stop_l \circ nil)\\
&\quad \parallel ((?c\_start_r \circ !tracert(hd, h) \circ \ldots) \ltimes^{\mathcal{I}_r} ?c\_stop_r \circ nil)\\
&\parallel_{\mathcal{L}} \Big(!c\_start_l \circ !c\_start_r \circ (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)\Big)\\
&\quad \text{with } \mathcal{I}_l = \{?stop_l\}, \mathcal{I}_r = \{?stop_r\}
\end{aligned}$$

The global test is associated a suitable environment by renaming the two verdict variables of $t_{extRelay(h)}$ and $t_{inDMZ(h)}$. The new environment contains the following variables: ver_1 for $t_{extRelay(h)}$, ver_2 for $t_{inDMZ(h)}$, ver_l, ver_r for $\mathsf{C}_{\mathcal{O}}$.

$t_{extRelay(h)}$, and $t_{inDMZ(h)}$ communicate with the controller through channels in $\mathcal{L}$.

6.2 $\mathcal{O}_T$-Rule

Security policies may also express availability requirements. Consider *"When there is a request to open an account, user privileges and resources must be activated within one hour"*. We formalize this requirement as:

$$request\_open\_account(c)[\neg ex\_account(c)] \Rightarrow \mathcal{O}_{1\mathrm{H}}(open\_account(c)[allocate\_disk(c)])$$

Supposing that there exists a tile for each predicate and that all tiles are independent. One could generate a test from appropriate derivation:

$$\begin{aligned}
&GenTest(req\_acc(c)[\neg ex\_acc(c)] \Rightarrow \mathcal{O}_{1H}(op\_acc(c)[alloc(c)]))\\
&\overset{\text{def}}{=} \big(GenTest_P(req\_acc(c)[\neg ex\_acc(c)], \mathcal{L}_l) \parallel GenTest_P(op\_acc(c)[alloc(c)], \mathcal{L}_r)\big)\\
&\quad \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}_{1H}}(\mathcal{L}_l, \mathcal{L}_r)\\
&= \Big(\big(Test(req\_acc(c), \mathcal{L}_{le}) \parallel GenTest_C(ex\_acc(c), \mathcal{L}_{lc})\big) \parallel_{\mathcal{L}_{le} \cup \mathcal{L}_{lc}} \mathsf{C}_E(\mathcal{L}_l, \mathcal{L}_{le}, \mathcal{L}_{lc})\Big)\\
&\parallel \Big(\big(Test(op\_acc(c), \mathcal{L}_{re}) \parallel GenTest_C(alloc\_disk(c), \mathcal{L}_{rc})\big) \parallel_{\mathcal{L}_{re} \cup \mathcal{L}_{rc}} \mathsf{C}_E(\mathcal{L}_r, \mathcal{L}_{re}, \mathcal{L}_{rc})\Big)\\
&\quad \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}_{1H}}(\mathcal{L}_l, \mathcal{L}_r)\\
&= \Big(\big(Test(req\_acc(c), \mathcal{L}_{le}) \parallel Test_C(ex\_acc(c), \mathcal{L}_{lc})\big) \parallel_{\mathcal{L}_{le} \cup \mathcal{L}_{lc}} \mathsf{C}_E(\mathcal{L}_l, \mathcal{L}_{le}, \mathcal{L}_{lc})\Big)\\
&\quad \parallel \Big(\big(Test(op\_acc(c), \mathcal{L}_{re}) \parallel Test_C(alloc\_disk(c), \mathcal{L}_{rc})\big) \parallel_{\mathcal{L}_{re} \cup \mathcal{L}_{rc}} \mathsf{C}_E(\mathcal{L}_r, \mathcal{L}_{re}, \mathcal{L}_{rc})\Big)\\
&\quad \parallel_{\mathcal{L}} \mathsf{C}_{\mathcal{O}_{1H}}(\mathcal{L}_l, \mathcal{L}_r)\\
&\text{with: } \mathcal{L} = \mathcal{L}_l \cup \mathcal{L}_r,\\
&\quad \mathcal{L}_s = \{c\_start_s, c\_stop_s, c\_loop_s, c\_ver_s\}_s, s \in \{l, r, le, lc, re, rc\}
\end{aligned}$$

7 Conclusion

We have proposed a test generation technique for testing the validity of a temporal logical formula on a system under test. The originality of this approach is to produce the tests by combinations of elementary test cases (called tiles), associated to each atomic predicates of the formula. These tiles are supposed to be provided by the system designer or a test expert, and, assuming they are correct, it can be proved that the whole test case obtained is sound. The practical interest of this approach is that it can be applied even if a formal specification of the system under test is not available, or if the test execution needs to mix several interface levels. A concrete example of such a situation is network security testing, where the security policy is usually expressed as a set of logical requirements, encompassing many network elements (communication protocols, firewalls, antivirus softwares, etc.) and those behavior would be hard to describe on a single formal specification. The abstract test cases we obtain are expressed in a process algebraic style, and they are structured into test drivers (the tiles), and test controllers (encoding the logical operators). This approach makes them close to executable test cases, and easy to map on a concrete (and distributed) test architecture. Independent parts of the tests can then be executed concurrently.

This work could be continued in several directions. First, the logic we proposed here could be extended. So far, the kind of formulae we considered was guided by a concrete application, but, staying in the context of network security, other deontic/temporal modalities could be foreseen, like"interdiction within a delay", or "permission". We also believe that this approach would be flexible enough to be used in other application domains, with other kinds of logical formulae (for instance with nested temporal modalities, which were not considered here). A second improvement would be to produce a clear diagnostic when a test execution fails. So far, test controllers only propagate "fail" verdicts, but it could be useful to better indicate to the user why a test execution failed (which subformula was unsuccessfully tested, and what is the incorrect execution sequence we obtained). Finally, we are currently implementing this test generation technique, and we expect that practical experimentations will help us to extend it towards the generation of *concrete* test cases, that could be directly executable.

References

1. ISO/IEC 9946-1: OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework. International Standard ISO/IEC 9646-1/2/3 (1992)
2. Brinksma, E., Alderden, R., Langerak, R. Van de Lagemaat, J., Tretmans, J.: A Formal Approach to Conformance Testing. In De Meer, J., Mackert, L., Effelsberg, W., eds.: Second International Workshop on Protocol Test Systems, North Holland (1990) 349–363
3. Tretmans, J.: Test Generation with Inputs, Outputs, and Quiescence. In Margaria, T., Steffen, B., eds.: Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96). Volume 1055 of Lecture Notes in Computer Science., Springer-Verlag (1996) 127–146
4. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. In: The Sixth World Conference on Integrated Design & Process Technology (IDPT'02), Pasadena, California, USA (2002)
5. Belinfante, A., Feenstra, J., de Vries, R., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal Test Automation : a Simple Experiment. In: 12th International Workshop on Testing of Communicating Systems, G. Csopaki et S. Dibuz et K. Tarnay, Kluwer Academic Publishers (1999)
6. Schmitt, M., Koch, B., Grabowski, J., Hogrefe, D.: Autolink - A Tool for Automatic and Semi-Automatic Test Generation from SDL Specifications. Technical Report A-98-05, Medical University of Lübeck (1998)
7. Groz, R., Jéron, T., Kerbrat, A.: Automated test generation from SDL specifications. In Dssouli, R., von Bochmann, G., Lahav, Y., eds.: SDL'99 The Next Millenium, 9th SDL Forum, Montreal, Quebec, Elsevier (1999) 135–152
8. Darmaillacq, V., Fernandez, J.C., Groz, R., Mounier, L., Richier, J.L.: Test Generation for Network Security Rules. In: 18th IFIP International Conference, TestCom 2006, New York, LNCS 3964, Springer (2006)
9. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1980)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)

A Complete definitions of controllers

A.1 The $\mathtt{C}_{\mathcal{O}}$ controller

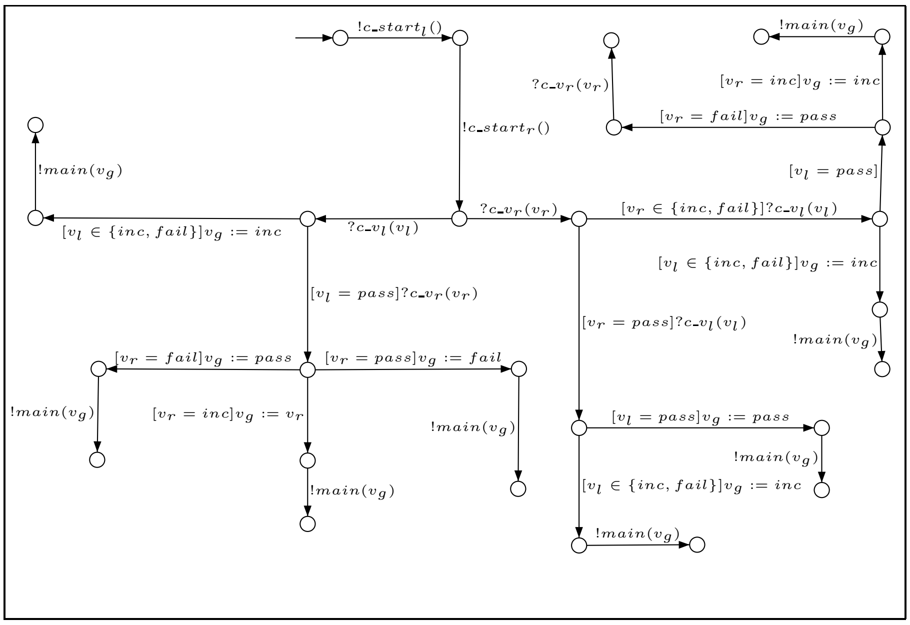
$$\mathtt{C}_{\mathcal{O}} \overset{\text{def}}{=} !c\_start_l \circ !c\_start_r \;\circ\; ((?c\_ver_r(v_r) \;\circ\; t_r) + (?c\_ver_l(v_l) \;\circ\; t_l))$$

$$\begin{aligned}
t_r &\overset{\text{def}}{=} ([v_r \in \{inc, fail\}]?c\_ver_l(v_l) \circ t_{r1}) + ([v_r = pass]?c\_ver_l(v_l))\\
t_l &\overset{\text{def}}{=} (([v_l \in \{inc, fail\}]v_g := inc) + ([v_l = pass]?c\_ver_r(v_r) \circ v_g := v_r)) \circ end\\
t_{r1} &\overset{\text{def}}{=} ([v_l = pass]v_g := v_r + [v_l \in \{inc, fail\}]v_g := inc) \circ end\\
t_{r2} &\overset{\text{def}}{=} ([v_l = pass]v_g := pass + [v_l \in \{inc, fail\}]v_g := inc) \circ end\\
end &\overset{\text{def}}{=} !main(v_g)
\end{aligned}$$

A.2 The $\mathtt{C}_{\mathcal{F}}$ controller

The automaton shown in Figure 7 presents the $\mathtt{C}_{\mathcal{F}}(\mathcal{L}_l, \mathcal{L}_r)$ controller,with:

$$\mathcal{L}_l = \{c\_start_l, c\_stop_l, c\_loop_l, c\_v_l\},$$
$$\mathcal{L}_r = \{c\_start_r, c\_stop_r, c\_loop_r, c\_v_r\})$$



$!c\_start_l()$
$!c\_start_r()$
$?c\_v_r(v_r)$
$!main(v_g)$
$[v_r = inc]v_g := inc$
$[v_r = fail]v_g := pass$
$[v_l = pass]$
$!main(v_g)$
$?c\_v_r(v_r)$
$[v_r \in \{inc, fail\}]?c\_v_l(v_l)$
$[v_l \in \{inc, fail\}]v_g := inc$
$?c\_v_l(v_l)$
$[v_l \in \{inc, fail\}]v_g := inc$
$[v_l = pass]?c\_v_r(v_r)$
$[v_r = pass]?c\_v_l(v_l)$
$!main(v_g)$
$[v_r = fail]v_g := pass$
$[v_r = pass]v_g := fail$
$!main(v_g)$
$[v_r = inc]v_g := v_r$
$!main(v_g)$
$[v_l = pass]v_g := pass$
$!main(v_g)$
$!main(v_g)$
$[v_l \in \{inc, fail\}]v_g := inc$
$!main(v_g)$

Fig. 7. An instanciated automaton representation of the $\mathtt{C}_{\mathcal{F}}$ controller.

A.3 The C_E controller

The automaton shown in Figure 8 presents the $\mathsf{C}_E(\mathcal{L}, \mathcal{L}_e, \mathcal{L}_c)$ controller, with:

$$\mathcal{L} = \{c\_start, c\_stop, c\_loop, c\_ver\},$$
$$\mathcal{L}_e = \{c\_start_e, c\_stop_e, c\_loop_e, c\_ver_e\},$$
$$\mathcal{L}_c = \{c\_start_c, c\_stop_c, c\_loop_c, c\_ver_c\}.$$

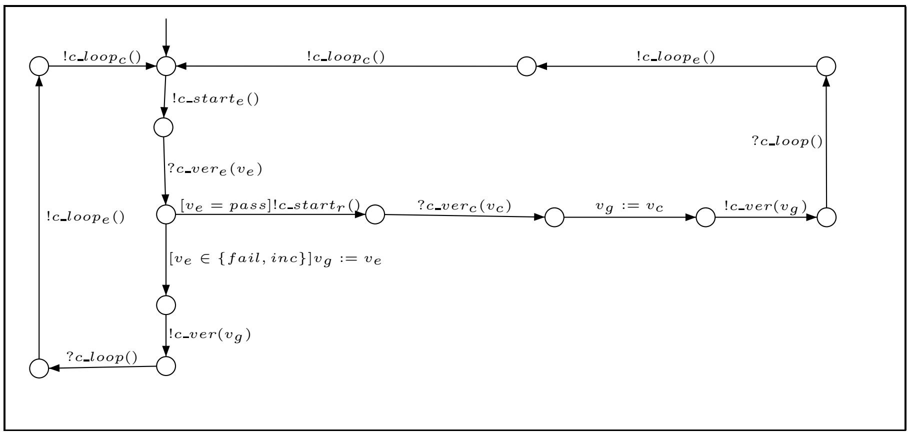


Fig. 8. An instanciated automaton representation of the C_E controller.

A.4 The $\mathsf{C}_\wedge$ controller

The automaton shown in Figure 9 presents the $\mathsf{C}_\wedge(\mathcal{L}, (\mathcal{L}_i)_{i=1\ldots n})$ controller, with:

$$\mathcal{L} = \{c\_start, c\_stop, c\_loop, c\_ver\},$$
$$\mathcal{L}_i = \{c\_start', c\_stop', c\_loop', c\_ver_i\}, i = 1 \ldots n.$$

B Execution trace of a $\mathcal{O}$-rule generated test

We show a possible execution of this process, the choice considered make $extRelay$ finishing first with $fail$. Then, the controller computes an inc verdict, and send it to the user through the channel $main$. We list here only the term rewritings, and show the evolution of the environment ρ only when it is modified.
We start from the test generated from $GenTest(extRelay(h) \Rightarrow \mathcal{O}\ inDMZ(h))$.
In the following, $\mathcal{I}_l = \{?c\_stop_l\}, \mathcal{I}_r = \{?c\_stop_r\}$.

$$\big((?c\_start_l \circ !connect(he, h) \circ \ldots) \ltimes^{\mathcal{I}_l} ?c\_stop_l \circ nil\big) \parallel$$
$$\big((?c\_start_r \circ !tracert(hd, h) \circ \ldots) \ltimes^{\mathcal{I}_r} ?c\_stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} \Big(!c\_start_l \circ !c\_start_r \circ (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)\Big)$$

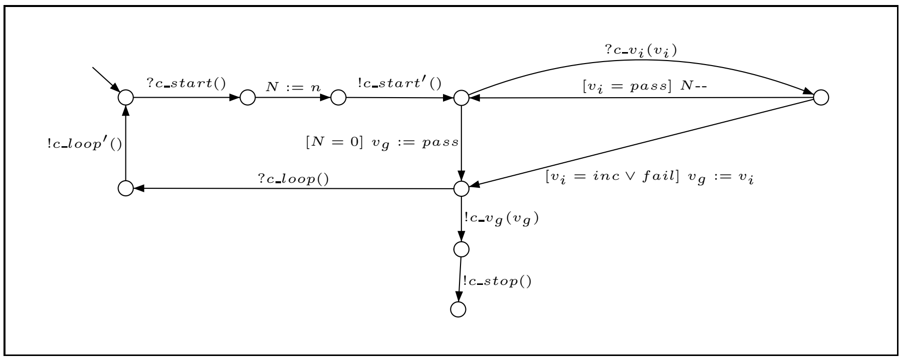


Fig. 9. An instanciated automaton representation of the $\mathsf{C}_\wedge$ controller.

$$\downarrow c\_start_l$$

$$\big((!connect(he,h) \circ \ldots) \ltimes^{\mathcal{I}_l} ?c\_stop_l \circ nil\big) \parallel$$
$$\big((?c\_start_r \circ !tracert(hd,h) \circ \ldots) \ltimes^{\mathcal{I}_r} ?c\_stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} \Big(!c\_start_r \circ (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)\Big)$$

$$\downarrow c\_start_r$$

$$\big((!connect(he,h) \circ \ldots) \ltimes^{\mathcal{I}_l} ?c\_stop_l \circ nil\big) \parallel \big((!tracert(hd,h) \circ \ldots) \ltimes^{\mathcal{I}_r} ?c\_stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)$$

$$\downarrow !connect(he,h)$$

$$\big((?ok \circ \ldots + ?ko \circ \ldots) \ltimes^{\mathcal{I}_l} ?stop_l \circ nil\big) \parallel \big((!tracert(hd,h) \circ \ldots) \ltimes^{\mathcal{I}_r} ?c\_stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots$$

$$\downarrow !tracert(hd,h)$$

$$\big((?ok \circ \ldots + ?ko \circ \ldots) \ltimes^{\mathcal{I}_l} ?stop_l \circ nil\big) \parallel \big((?end \circ \ldots + ?resp(hx) \circ \ldots) \ltimes^{\mathcal{I}_r} ?stop \circ nil\big)$$
$$\parallel_{\mathcal{L}} (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)$$

$$\downarrow ?ko$$

$$\big((ver_l := fail \circ !c\_v_l(ver_l) \circ nil) \ltimes^{\mathcal{I}_l} ?stop_l \circ nil\big) \parallel$$
$$\big(?end \circ \ldots + ?resp(hx) \circ \ldots) \ltimes^{\mathcal{I}_r} ?stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)$$

$$\downarrow ?resp(hx)$$

$$\big((ver_l := fail \circ !c\_v_l(ver_l) \circ nil) \ltimes^{\mathcal{I}_l} ?stop_l \circ nil\big) \parallel$$
$$\big((?resp(hx') \circ \ldots + ?end \circ \ldots) \ltimes^{\mathcal{I}_r} ?stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)$$

$$\downarrow ver_l := fail$$

$$(!c\_v_l(ver_l) \circ nil \ltimes^{\mathcal{I}_l} ?stop_l \circ nil) \parallel \big((?resp(hx') \circ \ldots + ?end \circ \ldots) \ltimes^{\mathcal{I}_r} ?stop_r \circ nil\big)$$
$$\parallel_{\mathcal{L}} (?c\_v_l(v_l) \circ \ldots + ?c\_v_r(v_r) \circ \ldots)$$
$$\rho = \rho_0[fail/ver_l]$$

$$\downarrow c\_v_l(fail)$$

$$(nil \ltimes^{\mathcal{I}_l} ?stop_l \circ nil) \parallel \big((?resp(hx') \circ \ldots + ?end \circ \ldots) \ltimes^{\mathcal{I}_r} ?stop_r \circ nil\big)$$

$$\|_{\mathcal{L}} \ ([v_l = fail \vee v_g := inc]/v_g := inc \circ \ldots + [v_1 = pass]/v_g := inc \circ \ldots)$$

$$\downarrow [v_l = fail \vee inc]/v_g := inc$$

$$(nil \ltimes^{\mathcal{I}_l} ?stop_l \circ nil) \parallel \big((?resp(hx') \circ \ldots + ?end \circ \ldots \ltimes^{\mathcal{I}_r} ?stop_r \circ nil\big))$$
$$\|_{\mathcal{L}} \ (!stop \circ !main(v_g) \circ nil)$$
$$\rho = \rho_0[fail/ver_l, inc/v_g]$$

$$\downarrow stop()$$

$$(nil \parallel nil) \|_{\mathcal{L}} \ (!main(v_g) \circ nil)$$
$$\rho = \rho_0[fail/ver_l, inc/v_g]$$

$$\downarrow main(inc)$$

$$(nil \parallel nil) \|_{\mathcal{L}} \ nil$$