ROBUSTNESS TESTING OF COMMUNICATING SYSTEMS : FRAMEWORK AND CASE STUDY

A. ROLLET, F. SAAD-KHORCHEF

Abstract. A system is considered as robust if it is able to adopt an acceptable behavior under unexpected running conditions. Nowadays, robustness testing is an important aspect of the validation process of complex software. The aim of such testing is to lead the system into unexpected conditions and to check the behavior. In this article, we propose a framework and a tool for robustness test cases generation. Our framework consists of two main phases. The first one constructs an *increased specification* by integrating hazards in the nominal specification model. The rule of the increased specification is to specify the acceptable behavior in presence of hazards. This phase uses a robustness relation permitting to check robustness of an IUT (Implementation Under Test) compared to the increased specification. The second phase provides a specific method to generate robustness test cases from the increased specification and a robustness test purpose. Our tool permits to increase the specification, and to generate automatically robustness test cases (in TTNC-3 format) from system specifications written in SDL. We also give some experimental results on the SSL handshake protocol.

Keywords: Robustness testing, Formal testing, Robustness relation, IOLTS, RTCG Tool, Communicating systems, SSL protocol

1. Introduction

In recent software or hardware development, formal validation is highly needed in order to reduce development cost and to avoid catastrophic errors. The complexity of present systems becomes higher and higher. Then, a proper validation is highly needed in order to increase the quality and the confidence of the system. Testing is an important part of the validation process consisting in a direct execution of the system implementation (IUT for *Implementation Under Test*) using a tester. Resulting outputs are observed and compared to the expected behavior of the system or component. Testing may focus on different topics such as conformance, reliability, interoperability and robustness. Existing generation techniques usually deal with conformance testing : the principle is to use a formal specification of the system in order to generate automatically

sequences. These sequences are applied on the IUT using the tester, and results are observed and compared with the specification.

In this article, we deal with robustness testing of communicating systems (e.g. communicating protocols). Although a precise definition of robustness is somewhat elusive, functionally the meaning is clear : "the ability of a system to function correctly in presence of faults or stressful environmental conditions" [*IEEE std 610.12-1990*] described in [11]. The term "hazards" will be used to gather faults and stressful conditions. Robustness is an important aspect of a software : many bugs are caused by a situation not expected in the specification : indeed system specifications usually do not take care about unexpected conditions. Note that it is never possible to have a complete specification of the system directly, but it is possible to specify a behavior facing a specific hazard when this latter is identified. Then the major problem of robustness testing is to find a way to lead the system into unexpected situations.

In the hardware domain robustness testing has been well studied, contrary to the software domain in which less contributions are available up to our knowledge. One contribution of our work is to provide a framework permitting to take into account robustness aspects. This aim is achieved by integrating representable hazards in the nominal specification of the system. The obtained model is called the *increased specification*. Because of its possibly important size, we propose a specific method to generate robustness test cases using a test purpose. We propose a generation method inspired by conformance testing technics.

We present the RTCG tool which implements the previous approach. Firstly, RTCG provides some help to obtain an increased specification. Then RTCG permits to extract robustness test cases (in the TTCN-3 format described in [19]) based on a given robustness test purpose and on the increased specification (written in the SDL format specified in [9]).

The article is organized as follows. We give a state of the art in section 2. Section 3 recalls the models used in our study. Section 4 presents our framework for robustness testing. Section 5 describes the RTCG tool and provides a case study on the SSL handshake protocol and we finally conclude in section 6.

2. Related work

Many research have been done in the domain of protocol testing . The majority of these works deals with conformance testing, normalized in [8]. An overview may be found in [10]. In this section, we focus particularly on robustness testing works.

In [4], authors proposes a study on robustness testing, focusing on hazard classification and some possible directions to handle the problem. Authors define the robustness notion as "the ability of a system to function acceptably in the presence of faults or stressful environmental conditions" and provide a state of the contributions in this domain.

In [14], authors present the PROTOS project in which they describe the system with a high level of abstraction and then to simulate abnormal inputs in the specification. It is mainly focused on the detection of vulnerabilities of a network software system. In this case, robustness is restricted to the notion of network security.

Some approaches are based on software fault injection :
The FIAT tool exposed in [2] modifies a processus binary image in memory. In [12], authors propose to apply randomly interruptions in the IUT, whereas the BALLISTA tool works on data unexpected modifications. This idea is explained in []. These approaches are based on integration of faults directly in the software implementation of the system, but do not care about interpretation of different behaviors.

Another approach consists in using model-based test generation. The main difficulty of such technics is to describe the hazards in the model. Many works consider such approach : see for example [16, 5, 13].

In [16], authors propose a first approach based on a refusal graph used to model hazards. Contrary to our method, it only deals with inopportune inputs, but not with invalid inputs. Moreover, our approach distinguishes between inputs and outputs in the model.

In [5], authors use a formal fault model in order to build a "mutant" specification. They use a fault model in order to add "fault" transitions in the specification. They define a robustness relation based on a robustness property. Contrary to our approach, they do not permit to integrate unexpected inputs in the model.

The results in [13] show how to use a degraded specification to model the behavior in case of critical situation, and integrate the hazards directly in the test sequences. A major difference between works described in [13] and this work is in the concept of robustness : we consider here that robustness implies conformance; the method described in [13] does not.

3. Preliminaries

In this section, we introduce the models and notations used throughout the article.

3.1. Models of specification

Usually, communicating softwares are specified in a dedicated language (SDL, LOTOS, UML, etc...). Such formalisms are based on Labelled Transition System (LTS) semantics. LTS distinguishes internal and visible actions. But in black-box testing, a distinction is often made between inputs and outputs. In this article we use the IOLTS model (Input Output Labelled Transition System).

Definition 1 (IOLTS)**.**
An $IOLTS$ (see [17]*) is a quadruplet $S = (Q, \Sigma, \rightarrow, q_0)$ such that :*

- *Q is a nonempty finite set of states, q_0 is the initial state,*
- *Σ is the alphabet of actions,*
- *$\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation.*

The alphabet Σ is partitioned into three sets $\Sigma = \Sigma_O \cup \Sigma_I \cup I$, where Σ_O is the output alphabet (an output is denoted by $!a$), Σ_I is the input alphabet (an input is denoted by $?a$) and I is the alphabet of internal actions (an internal action is denoted by τ). Usual notations are:

Notation	Meaning
$q \xrightarrow{a}$	$\exists\, q' \mid q \xrightarrow{a} q'$
$q \xrightarrow{\mu_1 \dots \mu_n} q'$	$\exists\, q_0 \dots q_n \mid q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} q_n = q'$
$q \overset{\varepsilon}{\Rightarrow} q'$	$q = q'$ or $q \xrightarrow{\tau_1 \dots \tau_n} q'$
$q \overset{a}{\Rightarrow} q'$	$\exists\, q_1, q_2 \mid q \overset{\varepsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\varepsilon}{\Rightarrow} q'$
$q \overset{a_1 \dots a_n}{\Rightarrow} q'$	$\exists\, q_0 \dots q_n \mid q = q_0 \overset{a_1}{\Rightarrow} q_1 \overset{a_2}{\Rightarrow} \dots \overset{a_n}{\Rightarrow} q_n = q'$
$q\ after\ \sigma$	$\{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$; by extension, $S\ after\ \sigma = q_0\ after\ \sigma$
$Trace(q)$	$\{\sigma \in \Sigma^* \mid q \overset{\sigma}{\Rightarrow}\}$; by extension, $Trace(S) = Trace(q_0)$
$Out(q)$	$\{a \in \Sigma_O \mid q \overset{a}{\Rightarrow}\}$
$Out(S, \sigma)$	$Out(S\ after\ \sigma)$
$ref(q)$	$\{a \in \Sigma_I \mid a \overset{a}{\not\rightarrow})\}$

The observable behavior is described by $\Rightarrow$. $q\,after\,\sigma$ is the set of reachable states from q by σ. $Trace(q)$ is the set of observable sequences starting from q. $Out(q)$ is the set of all possible outputs of q. $ref(q)$ is the set of inputs not specified in the state q. Σ^* is the language associated to S.

Example 3.1. *In Figure.1 (right) :*

- $\Sigma = \{?a, ?b, !x, !y\}$ *with* $\Sigma_I = \{?a, ?b\}$ *and* $\Sigma_O = \{!x, !y\}$,
- $Trace(q) = \{?a, ?a.!x, ?a.?b, ?a.?b.!y, ...\}$,
- *for* $\sigma = ?a.?b$, $q_0\ after\ \sigma = q_3$,
- $Out(q_0) = Out(q_2) = \emptyset$, $Out(q_1) = \{!x\}$ *and* $Out(q_3) = \{!y\}$,

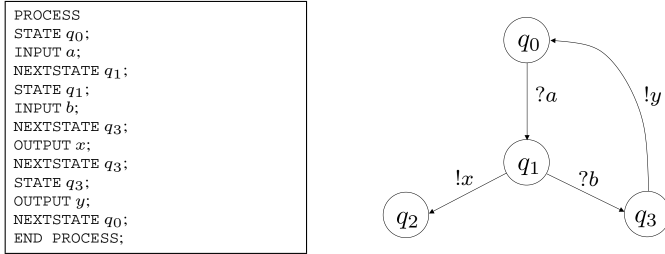

Figure 1: From SDL specification to IOLTS model

- *for* $\sigma = ?a.?b$, $Out(S, \sigma) = \{!y\}$,
- $ref(q_0) = \{?b\}$, $ref(q_1) = \{?a\}$ *and* $ref(q_2) = ref(q_3) = \{?a, ?b\}$

An $IOLTS$ S is called *deterministic* if no state accepts more than one successor with an observable action. It is called *observable* if no transition is labelled by τ. S is called *input-complete* if each state accepts all inputs of the alphabet. In Figure 1, S is deterministic, observable but not input-complete.

3.2. Hazards

In robustness testing, a *hazard* denotes any event not expected in the nominal specification of the system. They may be internal, external or beyond the system boundaries (the different notions are explained in [4]) or classified according to tester controllability or/and formal representability (as explained in [15]). In this article, we deal with controllable and representable hazards related to communicating software domain. Controllability means the ability of the tester to control the presence of hazards (e.g. erroneous or unexpected inputs), and representability means that it is possible to represent the hazard in the IOLTS model (e.g. inputs or outputs). More precisely, we identify three kinds of controllable and representable hazards :

3.2.1. Invalid Inputs

In a hostile environment, exchanged messages may be infected by accidental or intentional faults. Formally, we consider as an "*invalid input*" any unspecified input. i.e, $?a' \notin \Sigma_I$. In Figure.1 : let $?a'$ be a random mutation of $?a$. $?a'$ is considered as an invalid input.

3.2.2. Inopportune Inputs

In a hostile environment, the communicating software entity may receive delayed or untidy messages. Formally, "*inopportune inputs*" correspond to actions which exist in the alphabet of the specification, but not expected in the given state. $ref(q)$ (see standard notations of $IOLTS$) denotes the inopportune inputs in a state $q \in Q$. In state q_0 of Figure.1 : $?b$ is considered as an inopportune input.

3.2.3. Unexpected outputs

Taking into account the hazards can lead the system, in some cases, to send some unexpected outputs. Sometimes, such outputs may be considered as acceptable. For example, restarting a session, resetting or closing a connection may be acceptable behaviors. As a consequence, all acceptable outputs must be added to the specification (e.g. restarting or closing connection messages). Formally, $!x'$ is an unexpected output if $!x' \notin \Sigma_O$ or $!x' \in \Sigma_O \wedge !x' \notin Out(q)$.

4. Proposed approach

In this section, we outline our formal approach to generate robustness test cases. Two phases are given : firstly we construct an increased specification, and secondly we generate robustness test cases. Note that the nominal specification describes the expected behavior in nominal conditions. In the following, it is modelled by an IOLTS denoted S.

4.1. First phase : Increase of specification

This phase consists in integrating the representable hazards (invalid inputs, inopportune inputs and acceptable outputs) in the model of the nominal specification. The obtained model is called *increased specification*.

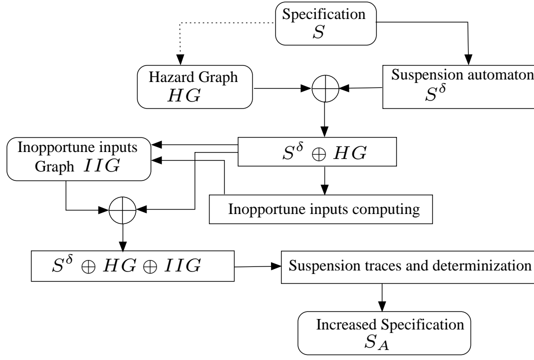

Figure 2: Obtaining the increased specification

The aim of the *increased specification* is to formally describe the acceptable behaviors in presence of controllable and representable hazards. Robustness of an implementation is evaluated with respect to the increased specification. The different steps in order to obtain the increased specification are summarized in Figure 2, and are detailed just below.

4.1.1. Quiescence

In practice, the tester observes outputs of a system, but also the absence of events (quiescence). Several kinds of quiescence may happen in a state $q \in Q$:

- ***outputlock*** quiescence if the system is blocked on standby input of the environment ($Out(q) = \emptyset$),
- ***deadlock*** quiescence if there is no more evolution of the system ($\forall a \in \Sigma | q \stackrel{a}{\not\to}$),
- ***livelock*** quiescence if $q \stackrel{\varepsilon}{\Rightarrow} q$.

To model valid quiescence in $IOLTS$ model, we use the suspension automaton defined below :

Definition 2 (Suspension automaton)**.** *The suspension automaton (see* [18]*) associated to $S = (Q, \Sigma, \to, q_0)$ is an IOLTS $S^\delta = (Q, \Sigma^\delta, \to_\delta, q_0)$ such that: $\Sigma^\delta = \Sigma \cup \{\delta\}$ with $\delta \in \Sigma_O^\delta$. $\to_\delta$ is obtained from $\to$ by adding loops $q \stackrel{\delta}{\to} q$ for all quiescence states.*

Thus, quiescence is seen as an observable output action. In practice, the tester identifies such event with a timeout.

Example 4.1. *In Figure 3.(b), q_0 is an outputlock quiescent state and q_2 is a deadlock quiescent state.*

The first step of our approach consists in obtaining the suspension automaton S^δ associated to S.

4.1.2. Acceptable behavior

In order to check the robustness of the system, the acceptable behavior in the presence of hazards has to be given by the system designers. The acceptable behavior is supposed modelled by a specific graph called *meta-graph*.

Let $S = (Q, q_0, \Sigma, \rightarrow_S)$ a nominal specification. A meta-graph G, associated to S, is a graph such that each state of G corresponds to a set of states of S having the same behaviors in the presence of the same hazards.

Definition 3. *A meta-graph associated to S is a triplet $G = (V, E, L)$ such as :*

- $V = V_d \cup V_m$ *is a set of states.* $V_m \subseteq 2^Q$ *is called the set of* meta-states *and* V_d *is called the set of* degraded states *such that* $V_d \cap Q = \emptyset$*.*
- L *is an alphabet of actions,*
- $E \subseteq V \times L \times V$ *is a set of edges.*

In the following, we suppose that invalid inputs and acceptable outputs are modelled by one or more meta-graph(s) HG (*Hazards Graph*), and inopportune inputs are represented by meta-graph(s) IIG (*Inopportune Input Graph*). Using two different types of meta-graph permits to firstly integrate invalid inputs, provided by the testers, in the specification model. Then, the new input set (invalid inputs and valid inputs) is used to compute the inopportune inputs.

4.1.3. Integrating hazards

This step consists in the composition of the nominal specification S and a hazard graph HG. The composition between an IOLTS and a meta-graph is defined by :

Definition 4 (Composition $IOLTS \oplus G$).
Let $S = (Q, q_0, \Sigma, \rightarrow_S)$ be an $IOLTS$ and $G = (V, E, L)$ a meta-graph associated to S. The composition of S and G, noted $S \oplus G$, is the $IOLTS$ $(Q^{S\oplus G}, q_0^{S\oplus G}, \Sigma^{S\oplus G}, \rightarrow_{S\oplus G})$ defined by: $Q^{S\oplus G} = Q \cup V_d$, $q_0^{S\oplus G} = q_0$, $\Sigma^{S\oplus G} = \Sigma \cup L$ and the following rules :

1. $q \xrightarrow{a} q' \iff q \xrightarrow{a}_{S\oplus G} q'$
2. $(v, a, v') \in E$ *and* $v, v' \in V_d \iff v \xrightarrow{a}_{S\oplus G} v'$.
3. $(v, a, v') \in E$, $v \in V_m$ *and* $v' \in V_d \iff \forall\, q \in v,\ q \xrightarrow{a}_{S\oplus G} v'$.
4. $(v, a, v') \in E$, $v \in V_d$ *and* $v' \in V_m \iff \forall\, q \in v',\ v \xrightarrow{a}_{S\oplus G} q$.
5. $(v, a, v') \in E$ *and* $v, v' \in V_m \iff \forall\, q \in v,\ q' \in v'$, $q \xrightarrow{a}_{S\oplus G} q'$.

6. $(v, a, v) \in E$ *and* $v \in V_m \iff \forall\, q \in v,\ q \xrightarrow{a}_{S \oplus G} q$.

This composition consists in adding in S the set of transitions and states of meta-graph HG. Actually, for a state q of S member of a meta-state (i.e. a set of states) v of HG, we add in S the set of transitions and/or states starting from v.

Example 4.2. *In Figure 3.(c), the compositions $S^\delta \oplus HG$ is obtained as follows :*
Rule 1 *adds to $S^\delta \oplus HG$ the whole of transitions of S^δ ($q_0 \xrightarrow{?a} q_1$, $q_1 \xrightarrow{!x} q_2$, $q_1 \xrightarrow{?b} q_3$, $q_3 \xrightarrow{!y} q_4$, $q_0 \xrightarrow{!\delta} q_0$, $q_2 \xrightarrow{!\delta} q_2$);*
Rule 2 *adds to $S^\delta \oplus HG$ the transition $d_2 \xrightarrow{?b'} d_1$;*
Rule 3 *adds to $S^\delta \oplus HG$ the following transitions ($q_0 \xrightarrow{?a'} d_2$, $q_1 \xrightarrow{?a'} d_2$, $q_2 \xrightarrow{?a'} d_2$, $q_3 \xrightarrow{?a'} d_2$, $q_0 \xrightarrow{!x'} d_1$, $q_1 \xrightarrow{!x'} d_1$, $q_2 \xrightarrow{!x'} d_1$, $q_3 \xrightarrow{!x'} d_1$);*
Rule 4 *adds to $S^\delta \oplus HG$ the following transitions ($d_1 \xrightarrow{?a} q_0$, $d_2 \xrightarrow{?a} q_0$);*
Rule 6 *adds to $S^\delta \oplus HG$ the following transitions ($q_0 \xrightarrow{?b'} q_0$, $q_1 \xrightarrow{?b'} q_1$, $q_2 \xrightarrow{?b'} q_2$, $q_3 \xrightarrow{?b'} q_3$).*
Rule 5 *is not used because there are no transitions between the meta-states.*

After the integration of invalid inputs and acceptable outputs in S^δ, we compute the inopportune inputs (using the ref set of each set) of $HG \oplus S^\delta$. It has to be done in a different step, since the increase of the alphabet is necessary before the inopportune inputs integration. Then, system designers give the required acceptable behavior in this case. The given description is modelled by IIG (Figure 3 (d)).
We reuse the definition 4 in order to integrate inopportune inputs in $HG \oplus S^\delta$. The obtained model is $HG \oplus S^\delta \oplus IIG$ (Figure 3 (e)).

4.1.4. Determinization

As robustness testing is based on the observation of visible behaviors, test synthesis requires a determinization of the specification. This means that two sequences of inputs always give the same sequence of outputs. Formally,

Definition 5 (Determinization of $IOLTS$)**.** *Let $S = (Q^S, q_0^S, \Sigma^S, \rightarrow_S)$ be an IOLTS. The deterministic $IOLTS$ obtained from S is denoted $\Delta(S)$ such as : $Traces(S) = Traces(\Delta(S))$.*
$\Delta(S) = (Q^{S^\Delta}, q_0^S\ after\ \varepsilon, \Sigma^{\Delta(S)}, \rightarrow_{\Delta(S)})$ *is defined as follow :*

- $Q^{S^\Delta} \subseteq 2^{Q^S}$*, i.e, some states of $\Delta(S)$ are the parts of Q^S,*

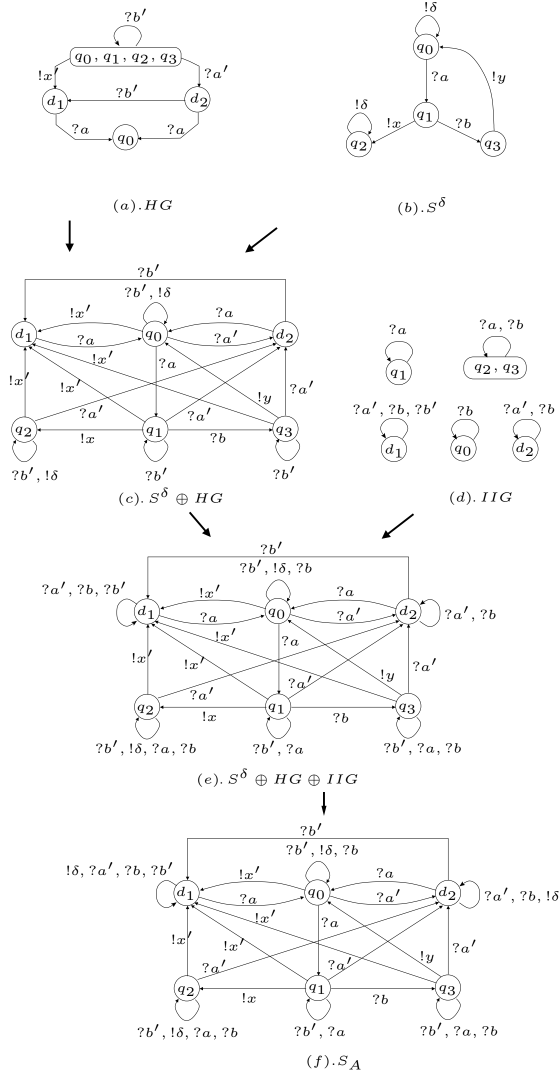

Figure 3: Construction of the increased specification

- *The initial state of* $\Delta(S)$ *is the state set of* S *reachable from* q_0^S *with internal actions. Formally,* $q_0^{\Delta(S)} = \{q_0^S\ after\ \varepsilon\}$,
- $\Sigma^{\Delta(S)} = \Sigma^S$ *is the set of observable actions,*

- $P \xrightarrow{a}_{\Delta(S)} P' \iff P, P' \in 2^{Q^S},\ a \in \Sigma^{\Delta(S)}$ *and* $P' = P\ after\ a$.

The deterministic model obtained from the suspension automaton associated to $HG \oplus S^\delta \oplus IIG$ is called the increased specification (Figure 3.(f)), and denoted S_A. It will be used as a base for the generation of robustness test cases.

4.1.5. Robustness relation

The IUT is a black box interacting with a tester. We apply the *test hypothesis* generally used in testing research, assuming that :

- IUT is modelled by an IOLTS $IUT = (Q^{IUT}, \Sigma^{IUT}, \rightarrow_{IUT}, {q_0}^{IUT})$ such that :
  ${\Sigma_I}^{S_A} \subseteq {\Sigma_I}^{IUT}$ and ${\Sigma_O}^{S_A} \subseteq {\Sigma_O}^{IUT}$;
- IUT is *input-complete* on the alphabet Σ^{S_A}.

We also assume that IUT conforms to S with respect to the conformance relation ***ioco*** (described in [17]). It is justified by the fact that, in our work, we consider that robustness of a system implies its conformance.

Let IUT be an implementation of a specification S and S_A its increased specification. The robustness relation **Robust** is defined by :

$$IUT\ \mathbf{Robust}\ S_A \equiv_{def} \forall \sigma \in Trace(S_A) \backslash Trace(S^\delta) \\ \iff Out(IUT^\delta, \sigma) \subseteq Out(S_A, \sigma).$$

Only the increased behaviors (added) are useful for robustness testing because the nominal behaviors (including valid quiescence) already passed the conformance testing.

Example 4.3. *Let us consider Figure 4.*

- IUT_1 ***Robust*** S_A *because all traces in* IUT_1 *are included in* S_A
- *not(*IUT_2 ***Robust*** S_A*) because* IUT_2 *after* $?a'$ *sends* $!y$ *but* S_A *after* $?a'$ *sends* $!x'$.

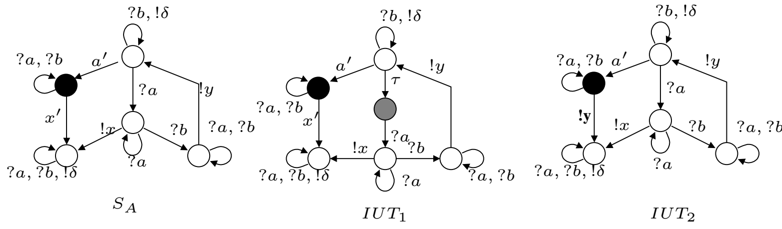

Figure 4: Robustness relation

4.2. Second phase : Robustness test generation

In this section we present a robustness test case generation technique. Using test purpose permits to reduce the test selection domain and to concentrate the efforts in order to check some critical functionalities. This phase may be summarized as follows :

1. Choice of robustness test purpose,
2. Synchronization between the specification and the test purpose in order to deduce the behaviors which satisfies the test purpose,
3. Constructing the *robustness test graph* from the mirror image (i.e. inputs become outputs and outputs become inputs) of the *synchronous product*
4. Constructing the *reduced robustness test graph* by deleting any trace rejected by the used robustness test purpose,
5. Extracting the *robustness test cases*.

The general view of this technique is given in Figure 5, and a detailed example is given in Figure 6.

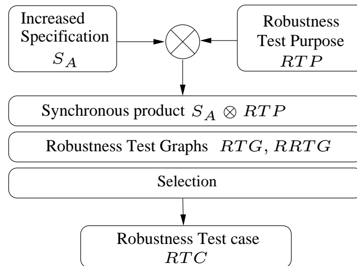

Figure 5: Robustness test cases generation

4.2.1. Robustness test purpose

A robustness test purpose (RTP) permits to select a part of the total specification in order to focus on a precise functionality (e.g, robustness property). Formally,

Definition 6 (RTP). *A robustness test purpose is a deterministic and observable IOLTS $RTP = (Q^{RTP}, \Sigma^{RTP}, \rightarrow_{RTP}, q_0^{RTP})$ with two sets of trap states "**Accept**" and "**Reject**", with the same alphabet as the increased specification (i.e. $\Sigma^{RTP} \subseteq \Sigma^{S_A}$) .*

Example 4.4. *The RTP given in Figure 6.(b) aims at seeking any trace of the increased specification containing a reception of the invalid input $?a'$ followed by the acceptable output $!x'$ without considering the transitions labelled by $!x$ or $?b$.*

The label "*other*" is used to describe all actions of the alphabet $\Sigma^{S_A \otimes RTP}$ which are not specified in the current state.

4.2.2. The synchronous product $S_A \otimes RTP$

In order to obtain a robustness test sequence, we have to cover simultaneously the RTP and S_A until we find an adequate sequence satisfying RTP. The synchronous product is defined as follows :

Definition 7 (Synchronous product). *Let $S_A = (Q^{S_A}, q_0^{S_A}, \Sigma^{S_A}, \rightarrow_{S_A})$ be an IOLTS of the increased specification, and $RTP = (Q^{RTP}, q_0^{RTP}, \Sigma^{RTP}, \rightarrow_{RTP})$ a robustness test purpose with $\Sigma^{RTP} = \Sigma^{S_A}$ and with state sets "**Accept**" and "**Reject**". The synchronous product of S_A and RTP, denoted by $S_A \otimes RTP$, is a deterministic $IOLTS$ $S_A \otimes RTP = (Q^{S_A \otimes RTP}, q_0^{S_A \otimes RTP}, \Sigma^{S_A \otimes RTP}$, $\rightarrow_{S_A \otimes RTP})$ defined by :*

1. $q_0^{S_A \otimes RTP} = (q_0^{S_A}, q_0^{RTP})$,
2. $Q^{S_A \otimes RTP} = \{(q_1, q_2) \mid q_1 \in Q^{S_A}, q_2 \in Q^{RTP}\}$,
3. $\Sigma^{S_A \otimes RTP} \subset \Sigma^{S_A} \cup \Sigma^{RTP} = \Sigma^{S_A}$,
4. $\rightarrow_{S_A \otimes RTP}$ *is defined by :*
   $(q, q') \in Q^{S_A \otimes RTP}, q \xrightarrow{a}_{S_A} q_1 \wedge q' \xrightarrow{a}_{RTP} q'_1 \Longleftrightarrow (q, q') \xrightarrow{a}_{S_A \otimes RTP} (q_1, q'_1)$.

4.2.3. Robustness test graphs

A robustness test graph (RTG) describes all tests corresponding to a given RTP. Formally, a RTG is a deterministic IOLTS $RTG = (Q^{RTG}, \Sigma^{RTG}, \rightarrow_{RTG},$

q_0^{RTG}), composed by three subsets of states **ACCEPT**, **REJECT** and **INCONC** such that :

- $\Sigma^{RTG} = {\Sigma_O}^{RTG} \cup {\Sigma_I}^{RTG}$ with ${\Sigma_I}^{RTG} = {\Sigma_O}^{S_A \otimes RTP}$ and ${\Sigma_O}^{RTG} = {\Sigma_I}^{S_A \otimes RTP}$ (mirror image);
- $Q^{RTG} =$ **ACCEPT** $\cup$ **REJECT** $\cup$ **INCONC** with
  1. **ACCEPT** $= \{q \in Q^{S_A \otimes RTP} \mid \exists \sigma \in \Sigma^{S_A \otimes RTP*}\ ,\ q \xrightarrow{\sigma}$ **Accept**$\}$ **ACCEPT** consists of states from which the state **Accept** is reachable,
  2. **INCONC** $= \{q' \in Q^{S_A \otimes RTP} \mid \exists q \in$ **ACCEPT**$, q' \notin$ **ACCEPT**$, a \in {\Sigma_O}^{S_A \otimes RTP}\ ,\ q \xrightarrow{a} q'\}$. i.e. **INCONC** is composed of states not in **ACCEPT**, but which are direct successors of states in **ACCEPT** by an output in $S_A \otimes RTP$,
  3. **REJECT** $= \{q \in Q^{S_A \otimes RTP} \mid q \notin$ **ACCEPT** $\wedge\ q \notin$ **INCONC**$\}$.
- if ${q_0}^{S_A \otimes RTP} \in$ **ACCEPT** then ${q_0}^{RTG} = {q_0}^{S_A \otimes RTP}$, otherwise Q^{RTG} is empty.

Since RTG is often voluminous, it is necessary to reduce it by concentrating only on the behaviors accepted by RTP. Then we keep in RTG only the paths leading to an ACCEPT or INCONC states. The obtained model is called reduced robustness test graph, and denoted by RRTG.

Example 4.5. *Robustness test graph RTG (Figure 6.(d)) describes the mirror image of the synchronous product (Figure 6.(c)). RTG consists of three states :* ***INCONC*** $= \{(q_2,$ ***Reject***$)\}$, ***REJECT*** $= \{(d_1,$ ***Reject***$)$, $(q_1,$ ***Reject***$)$, $(q_0,$ ***Reject***$)\}$ *and* ***ACCEPT*** $= \{\ (q_0, q_0')$, (d_1, q_1'), (q_1, q_0'), $(q_0,$ ***Accept***$)\ \}$. *Reduced robustness test graph RRTG (Figure 6.(e)) consists of the states and transitions of* ***ACCEPT*** *and* ***INCONC****.*

4.2.4. Robustness test case

A robustness test case (RTC) is an elementary test corresponding to a particular robustness test purpose. It describes the interactions between a tester and an implementation. It only contains observable actions.

Definition 8. *A robustness test case RTC is an $IOLTS$ $RTC = (Q^{RTC}, \Sigma^{RTC}, \rightarrow_{RTC}, q_0^{RTC})$ with three sets of trap states* ***Pass***, ***Fail*** *and* ***Inconc*** *characterizing verdicts. Its alphabet is $\Sigma^{RTC} = \Sigma_I^{RTC} \cup \Sigma_O^{RTC}$ with $\Sigma_O^{RTC} \subseteq \Sigma_I^{S_A}$ (RTC emits only inputs of S_A) and $\Sigma_I^{RTC} \subseteq \Sigma_O^{IUT}$ (RTC foresees any output or quiescence of IUT). We make several structural assumptions on test cases :*

- *states* ***Fail*** *and* ***Inconc*** *are directly reachable by inputs. Formally,* $\forall (q, a, q') \in \rightarrow_{RTC}\ \ (q \in$ ***Inconc*** $\cup$ ***Fail*** $\implies\ a \in {\Sigma_I}^{RTC})$,

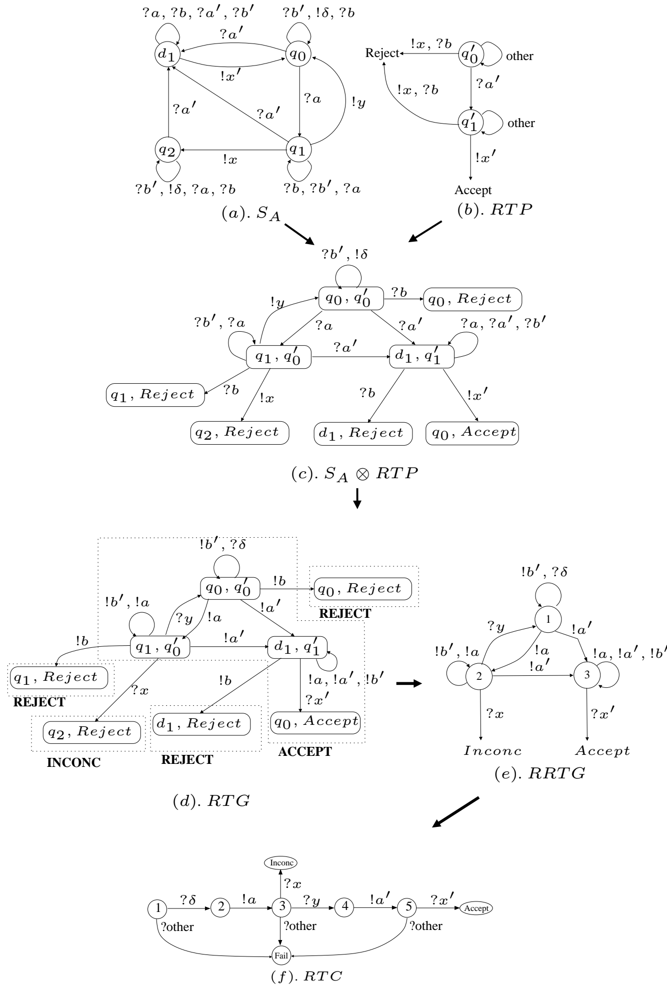

Figure 6: Robustness test cases generation

- *from each state a verdict must be reachable. Formally,* $\forall q \in Q^{RTC}, \exists \sigma \in \Sigma^{RTC*}, \exists q' \in$ ***Pass*** $\cup$ ***Fail*** $\cup$ ***Inconc***, $q \xrightarrow{\sigma} q'$,
- *RTC is controllable : no choice is allowed between two outputs or an input and output. Formally,* $\forall q \in Q^{RTC} \forall a \in {\Sigma_O}^{RTC}, q \xrightarrow{a}_{RTC} \implies \forall b \neq a, q \not\xrightarrow{b}_{RTC}$,

- *a test case is input complete in all states where an input is possible. Formally,*
  $\forall q \in Q^{RTC}(\exists a \in {\Sigma_I}^{RTC}, q \xrightarrow{a}_{RTC} \implies \forall b \in {\Sigma_I}^{RTC}, q \xrightarrow{b}_{RTC})$.

4.2.5. Selection of robustness test cases

In order to choose the traces which are considered in the robustness relation **Robust**, we use an algorithm based on coloration principle. Two colors distinguish the transitions of the nominal specification (first color) and those added during the construction of the increased specification (second color). Then we choose test cases favoring the second color (focusing on hazards), and we avoid any nominal trace (colored with the first color).
Algorithm 1 permits the selection of a random robustness test case. For all visited states in RRTG, we choose only one sending or all receptions until the **Accept** state is reached. Finally, we reject any RTC colored only with the same color as the nominal specification.

Algorithm 1 Computing of RTC

Require: Reduced Robustness Test Graph $RRTG$
Ensure: Robustness Test Case RTC
  repeat
    $Q^{RTC} := q_0$;
    for all not visited state q in Q^{RTC} **do**
      if q is a reception state **then**
        Add all started reception from q to RTC;
        Add all successor states of q by reception to Q^{RTC};
        Add the transition $q \xrightarrow{other} Fail$ to RTC;
      end if
      if q is sending/reception state **then**
        Random choice between sending or reception;
        if sending **then**
          Choice a random sending;
          Add the successor state of q by the selected sending to Q^{RTC};
        else
          Add all started reception from q to RTC;
          Add all successor states of q by reception to Q^{RTC};
          Add the transition $q \xrightarrow{other} Fail$ to RTC;
        end if
      end if
    end for
  until the RTC and the nominal specification colors are different.

Example 4.6. *Robustness test case (RTC) given in Figure 6.(f) is derived from RRTG (Figure 6.(e)). RTC consists of two output states (states 2 and 4), and three reception states (states 1, 3 and 5).*

5. Implementation and case study

5.1. RTCG tool

$RTCG$ (*Robustness Test Cases Generator*) is a tool automating the previous approach. It provides two functionalities :

The first one permits to build the increased specification of systems written in SDL or directly modelled as $IOLTS$. In order to achieve this aim, $RTCG$ implements the composition method given in paragraph 4.1.3. It computes the composition of the nominal specification S with a meta-graph HG. Then, it computes inopportune inputs from the previous product and, proposes a default (loops labeled with inopportune inputs in all states) increase or a customized increase. It also computes the suspension traces and the determinization.

The second one allows to generate robustness test cases based on a robustness test purpose. More precisely, the user defines both S_A and RTP files. $RTCG$ checks the RTP (observability, determinism and accept states). Then, $RTCG$ computes the synchronous product, the robustness test graph (RTG) and the reduced robustness test graph (RRTG). Finally it selects a robustness test case (RTC).

In the current $RTCG$ version, robustness test purposes and specification files are written in the SDL (specified in [9]) or DOT (see [1]) format and robustness test cases are written using the TTCN-3 (normalized in [7]), XML or DOT formats.

5.1.1. Case study : SSL protocol

In [6], authors describe SSL as follows : "The SSL protocol is designed to provide privacy between two communicating applications (a client and a server). Moreover, the protocol is designed to authenticate the server, and optionally the client". SSL is standardized by the IETF (Internet Engineering Task Force). The full specification of the SSL protocol is written in the RFC 2246. The SSL Protocol contains four under-protocols: *Handshake protocol*, *SSL Changes Cipher Spec protocol*, *SSL Alert protocol* and *SSL Record protocol*. The Handshake protocol is composed of two phases. First step deals with the selection of a cipher, the exchange of a master key and the authentication of the server. Second step handles client authentication if requested and finishes the handshaking. After the handshake stage is complete, the data transfer between client and server begins. All messages during handshaking and after are sent over the SSL Record protocol Layer.

Here, we deal only with the specification of the handshake protocol which describes three scenarios of communication as shown in figure 7.

```
PROCESS SSL-Hanshake (1,1);
START NEXTSTATE 1;
STATE 1;
 OUTPUT Client-Hello(no-sid);
 NEXTSTATE 2;
 OUTPUT  Client-Hello(sid);
 NEXTSTATE 3;
STATE 2;
 INPUT No-Certificate-Error;
 NEXTSTATE 4;
 INPUT ?Server-Hello(No-Hit);
 NEXTSTATE 5;
STATE 3;
 INPUT Server-Hello(No-Hit);
 NEXTSTATE 5;
 INPUT Server-Hello(Hit);
 NEXTSTATE 6;
STATE 4;
 INPUT Close-Connection;
 NEXTSTATE 1;
STATE 5;
 OUTPUT Bad-Certificate-Error;
 NEXTSATE 7;
 OUTPUT No-Cipher-Error;
 NEXTSTATE 7;
 OUTPUT Unsupported-Certificate-
        Type-Error;
 NEXTSTATE 7;
 OUTPUT Client-Master-Key;
 NEXTSTATE 6;
STATE 6;
 OUTPUT Client-Finished;
 NEXTSTATE 8;
 INPUT Server-Verify;
 NEXTSTATE 9;
STATE 7;
 OUTPUT Close-Connection;
 NEXTSTATE 1;
STATE 8;
 INPUT Server-Verify;
 NEXTSTATE 10;
STATE 9;
 OUTPUT Client-Finished;
 NEXTSTATE 10;
 INPUT Server-Finished;
 NEXTSTATE 11;
 INPUT Server-Request-Certificate;
 NEXTSTATE 12;
STATE 10;
 INPUT Server-Finished;
 NEXTSTATE 13;
 INPUT Server-Request-Certificate;
 NEXTSTATE 14;
STATE 11;
 OUTPUT Client-Finished;
 NEXTSTATE 14;
STATE 12;
 OUTPUT No-Certificate-Error;
 NEXTSTATE 15;
 OUTPUT  Client-Finished;
 NEXTSTATE 13;
 OUTPUT Client-Certificate;
 NEXTSTATE 17;
STATE 13;
 OUTPUT No-Certificate-Error;
 NEXTSTATE 16;
 OUTPUT Client-Certificate;
 NEXTSTATE 18;
STATE 14;
 INPUT SSL-Data-Record;
 NEXTSTATE 14;
 OUTPUT SSL-Data-Record;
 NEXTSTATE 14;
 INPUT Close-Connection;
 NEXTSTATE 1;
 OUTPUT Close-Connection;
 NEXTSTATE 1;
STATE 15;
 OUTPUT Client-Finished;
 NEXTSTATE 16;
 INPUT Server-Finished;
 NEXTSTATE 11;
 INPUT Close-Connection;
 NEXTSTATE 1;
STATE 16;
 INPUT Server-Finished;
 NEXTSTATE 14;
 INPUT Close-Connection;
 NEXTSTATE 1;
STATE 17;
 INPUT Server-Finished;
 NEXTSTATE 11;
 INPUT Bad-Certificate-Error;
 NEXTSTATE 19;
 INPUT Unsupported-Certificate-
       Type-Error;
 NEXTSTATE 19;
 OUTPUT Client-Finished;
 NEXTSTATE 18;
STATE 18;
 INPUT Bad-Certificate-Error;
 NEXTSTATE 20;
 INPUT Unsupported-Certificate-
       Type-Error;
 NEXTSTATE 20;
 INPUT Server-Finished;
 NEXTSTATE 14;
STATE 19;
 OUTPUT Client-Finished;
 NEXTSTATE 20;
 INPUT Close-Connection;
 NEXTSTATE 1;
 INPUT Server-Finished;
 NEXTSTATE 11;
STATE 20;
 INPUT Server-Finished;
 NEXTSTATE 14;
 INPUT Close-Connection;
 NEXTSTATE 1;
END PROCESS SSL-Hanshake;
```

Figure 7: SDL specification of the SSL Handshake protocol

The standard specification (RFC2246) defines the following errors :

- *No-Cipher-Error*. This error is returned by the client to the server when it can not find a cipher or key size. This error is not recoverable.
- *No-Certificate-Error*. When a *Request-Certificate* message is sent, this error may be returned if the client has no certificate to reply with. This error is recoverable (for client authentication only).
- *Bad-Certificate-Error*. This error is returned when a certificate is deemed bad by the receiving party. Bad means that either the signature of the certificate was bad or that the values in the certificate were inappropriate (e.g. a name in the certificate did not match the expected name). This error is recoverable (for client authentication only).
- *Unsupported-Certificat-Type-Error*. This error is returned when a client/ server receives a certificate type that it can not support. This error is recoverable (for client authentication only).

In [3], authors show that two error messages have been omitted in the reference document. The first, an *Unsupported-Authentication-Type-Error* message, is a mistake which would prevent the protocol using different methods of authentication of a client. The second, an *Unexpected-Message-Error* would allow an implementation to close the connection cleanly if an implementation sent an out-off-order message.

In order to verify the robustness of the Handshake protocol, we increase the nominal specification by integrating hazards (*invalid inputs* and *inopportune inputs*). Besides, to model the previous hazards, we consider the following hypothesis :

- if the implementation receives an invalid input then it closes the connection
- if it receives an inopportune input then it loops in the same state.

Formally, the previous hypothesis may be modelled by meta-graphs. The IOLTS of the increased specification is obtained from the SDL specification given in figure 7 and the previous robustness hypothesis. It is composed of 20 states and 176 transitions.

5.2. Robustness test generation with RTCG tool

In order to generate robustness test cases, we have defined a set of robustness test purposes aiming at checking the behavior of an implementation in presence of two invalid inputs (*Unexpected-Message-Error* and *Unsupported-Authentification-Type-Error*) and all inopportune inputs computed in each state :

1. RTP1 deals with the exchange message suite if no session identifier and no client authentication in the presence of the considered hazards : (*!client-hello, ?server-hello, !client-master-key, !client-finished, ?server-verify, ?server-finished"*) ;
2. RTP2 deals with the exchange message suite if session identifier and no client authentication in the presence of hazards : (*!client-hello, ?server-hello, !client-finished, ?server-verify, ?server-finished*);
3. RTP3 deals with the exchange message suite if session identifier and client authentication the the presence of the considered hazards : (*!client-hello, ?server-hello, !client-finished, ?server-verify, !server-request -certificate, ?client-certificate, ?server-finished*) ;

These RTPs consider only one occurrence of each hazard. In addition, we mention that both inopportune inputs and suspension traces are automatically generated by the RTCG tool. A corresponding test case is given in figure 8.

The increased specification is automatically computed from the nominal specification and the hazards graph. RTCG also applies the default increase of inopportune inputs. In figure 9, we give the results obtained with RTCG using these test purposes (WindowsXP ©, Pentium(R)4 CPU 2.80 GHz, RAM 256 Mo). In the figure, the “length” is the number of actions and RTC means “Robustness Test Case”. We compare them with the conformance testing tool TGSE in the same conditions. Test cases obtained with RTCG are significantly computed faster and test cases are usually shorter : RTCG focuses on transitions with hazards, avoiding pathes not useful for robustness testing.

	Property	RTC length	CPU Time (ms)
RTCG	RTP1	11	1.7
	RTP2	14	2.6
	RTP3	19	4.4
TGSE	RTP1	53	961.8
	RTP2	10	359.9
	RTP3	20	157.9

Figure 9: Results obtained with RTCG and TGSE

6. Concluding remarks

This article presented a framework and a tool permitting to generate robustness test cases for communicating software. The proposed approach consists of two phases : the first one deals with the construction of an increased specification. The second phase deals with robustness test cases generation. The tool

```
tescase Tester() runs on IUT {
timer ReponseTimer := 100E-3 ;
  Tester.send(No-Certificate-Error);
  ReponseTimer.start
  alt
      [] ReponseTimer.timeout
         {  setverdict(fail);
            stop
         }
      [] Tester.receive(Client-Hello(no-sid));
         {  setverdict(pass);
            ReponseTimer.stop
        Tester.send(Bad-Certificate-Error);
        Tester.send(Server-Hello(No-Hit));
        Tester.send(No-Certificate-Error);
        ReponseTimer.start
        alt
            [] ReponseTimer.timeout
               {  setverdict(fail);
                  stop
               }
            [] Tester.receive(Client-Master-Key);
               {  setverdict(pass);
                  ReponseTimer.stop
              Tester.send(Close-Connection);
              ReponseTimer.start
              alt
                  [] ReponseTimer.timeout
                     {  setverdict(fail);
                        stop
                     }
                  [] Tester.receive(Client-Finished);
                     {  setverdict(pass);
                        ReponseTimer.stop
                  [else] { setverdict(fail);
                           stop
                          }
                  }
            [else] { setverdict(fail);
                     stop
                   }
            }
      [else] { setverdict(fail);
               stop
             }
      }
control
  {
     execute (Tester());
  }
}
```

Figure 8: Test case for the SSL Handshake protocol

permits to implement the approach described above using SDL specification, and generating TTCN-3 test cases. We also proposed a case study on the SSL Handshake protocol.

This work is based on formal technics, and permits to take care about specific hazards when these latter are identified. It extends conformance testing technics to the problem of robustness testing.

As a future work, we intend to focus on unrepresentable hazards, and on models with time and data. The main difficulty in this case, is that test cases may become infinite. Then a symbolic approach could solve the problem.

References

[1] The DOT language. http://www.graphviz.org/doc/info/lang.html.

[2] J.-H. Barton, E.-W. Czeck, Z.-Z. Segall, and D.-P. Siewiorek. Fault injection experiments using FIAT. *IEEE Trans. Comput.*, 39(4):575–582, 1990.

[3] J. Bradley and N. Davies. Analysis of the SSL protocol. Technical Report CSTR-95-021, Department of Computer Science, University of Bristol, June 1995.

[4] R. CASTANET and H. WAESELYNK. Techniques avancées de test de systèmes complexes: Test de robustesse. Technical report, Action spécifique 23 du CNRS, 11 2003.

[5] J-C. FERNANDEZ, L. MOUNIER, and C. PACHON. A model-based approach for robustness testing. In LNCS, editor, *Testing of Communication Systems*, volume 3502, pages 333–348. ifip, may/june 2005.

[6] Kipp Hickman. The SSL protocol. Technical report, Netscape Communications Corp., Feb 9 1995.

[7] IEEE. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*, number 9646 in 2, 2003.

[8] IEEE. *International Organization for Standardization, Conformance testing methodology and framework - part 2: abstract test suite specification*, 2004.

[9] ITU-T. Specification and description language (sdl). ITU-T Recommendation no Z.105, International Telecommunication Union. Genève, 1999.

[10] T. JERON. Génération de tests pour les systèmes réactifs. un survol des théories et techniques. In IRIT, editor, *ETR2003. Systèmes, Réseaux et Applications*, pages 105–122. IRIT, Septembre 2003.

[11] IEEE Standard Glossary of Software Engineering Terminology 610.12-1990. Customer and terminology standards. *In IEEE Standards Software Engineering, IEEE Press*, 1, 1999.

[12] J. Regehr. Random testing of interrupt-driven software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298, New York, NY, USA, 2005. ACM Press.

[13] A. Rollet. Testing robustness of real-time embedded systems. *In Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of Formal Methods (FM 2003) Symposium, Pisa, Italy*, September 13 2003.

[14] J. Röning, M. Laakso, and A. Takanen. PROTOS - systematic approach to eliminate software vulnerabilities. http://www.ee.oulu.fi/research/ouspg, May 2002. 2002.

[15] F. Saad-khorchef, I. Berrada, A. Rollet, and R. Castanet. Automated robustness testing for reactive systems : Application to communicating protocols. In *6th International Workshop on Innovative Internet Community Systems (I2CS 2006), Neuchâtel, Switzerland*, June 26-28 2006.

[16] F. Saad-Khorchef and X. Delord. Une méthode pour le test de robustesse adaptée aux protocoles de communication. In *11ème Colloque Francophone sur l'Ingénierie des Protocoles CFIP'2005*, march 2005.

[17] J. TRETMANS. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[18] J. TRETMANS. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[19] ITU-T Recommendations Z.140-142. The testing and test control notation", version 3 (ttcn-3), rec. z.140: Ttcn-3 core language, rec. z.141: Tabular presentation format for ttcn-3 (tft), rec. z.142: Graphical presentation format for ttcn-3 (gft). ITU-T, Geneva (Switzerland), 2002.

Authors addresses:
LABRI - CNRS UMR 5800
Université Bordeaux 1 / ENSEIRB
351, cours de la Libération
F-33405 Talence
{rollet, saad-kho}@labri.fr