Testing robustness of communicating systems using ioco-based approach

Antoine Rollet
LABRI - CNRS (UMR 5800)
University of Bordeaux
33405 Talence cedex, France
rollet@labri.fr

Sébastien Salva
LIMOS - CNRS (UMR 6158)
Campus des Cézeaux
Aubière, France
salva@iut.u-clermont1.fr

Abstract

This paper deals with communicating system robustness testing by proposing a method for checking the correct behaviour of the Implementation Under Test (IUT) in unexpected situations. This formal method takes specifications written with IOLTS and generates robustness test cases by using the ioco theory. IOLTS are used to model many systems like distributed ones or Web service compositions. We present an algorithm permitting to generate sound test cases especially focusing on robustness aspects.

Key-words *: Robustness testing, Formal testing, Labelled Transition Systems, Hazards, ioco.*

1 Introduction

Software development companies are more and more aware that validation is a required step in software life cycle, especially since quality processes, like the CMMI (Capability Maturity Model Integration) model, are taken into consideration. Besides, using formal and automatic methods for test generation usually increases the confidence level of the system while reducing the total cost compared to manual and empiric ones.

It is very difficult to say that a system has the highest level of quality if this one is not robust. Robustness implies that the system will behave correctly despite unspecified events. This is why testing the system robustness is important. And in this context we propose a formal test cases generation method focusing on robustness aspects. We consider testing as "dynamic testing", i.e. testing consists in applying test sequences in the Implementation Under Test (IUT). Two main categories of testing are considered in the literature : (1) Functional testing : the source code is unknown. Only functional aspects are verified, using a specification (or the requirements) to generate tests. (2) Structural testing : the source code is known. Generation is done using the source code, usually focusing on code covering aspects.

In this work we propose a new method for robustness test cases generation of communicating systems. We only deal with black-box (i.e. the only information available from the specification are inputs and outputs) functional testing. We consider that the system is described with an Input Output Labeled Transition System (IOLTS) and we use it as a reference for test case generation. IOLTS may be used to specify many communicating systems like distributed ones or Web service compositions ([FTdV06]).

In [oSET99], robustness is defined as "the ability of a system to function correctly in presence of faults or stressful environmental conditions". Thus, we consider that testing the robustness means that we intend to focus on unexpected aspects of the system. Such unexpected events are usually called *hazards*. A study on hazards and their classification can be found in [SKRC07] and [CW03]. In this paper, we deal with external hazards. We consider three situations : (1) Occurrence of an input known by the system but not expected at this moment (2) Occurrence of an unknown input (not in the alphabet). (3) Occurrence of an unexpected output (in the alphabet or not).

The steps of our methodology are : (1) Computing the suspension automaton (2) Determinisation (3) Completion (4) Test case generation The relation used to check the correctness of the IUT in the **ioco** preorder ([TRE96]). Using this relation, we propose a complete test cases generation mainly focusing on robustness aspects, i.e. on hazards.

This article is structured as follows. In section 2, we present the definitions and models used in this paper. Section 3 mainly gives the contribution of this paper by presenting test cases generation method. We propose a brief discussion in Section 4. Section 5 contains related works on robustness testing, and finally we conclude in Section 6.

2 Models and definitions

In this work, systems are modeled with IOLTS (Input Output Labelled Transition System). We use this model to describe the specification, the implementation and the test cases. It is an extension of the LTS model permitting to distinguish the inputs (controllable) and the outputs (only observable) ([TRE96]). This model also allows to use internal transitions.

Definition 2.1 (IOLTS) *An Input Output Labelled Transition System (IOLTS) is a 4-uplet $S =< Q^S, q_0^S, \Sigma^S, \rightarrow_s>$ such that :(1) Q^S is a countable set of states; (2) $q_0^S \in Q$ is the initial state; (3) Σ^S is a countable alphabet of actions, partitioned into two subsets such that $\Sigma^S = \Sigma_i^S \cup \Sigma_o^S$ (respectively input marked with a $?$ and outputs marked with a !); (4) $\rightarrow_s \subset Q^S \times \Sigma_\tau^S \times Q^S$ is a set of transitions (Σ_τ^S means $\Sigma^S \cup \{\tau\}$ where τ is an internal action). Sometimes when it is obvious, ^{S} is not given in the notations.*

In order to analyse traces of the system, we need some further usual notations. We suppose that q, q', q_1, ..., $q_n \in Q^S$, a, a_1, ..., $a_n \in \Sigma^S$ and $\sigma \in (\Sigma^S)^*$.

- $q \xrightarrow{a}$ means $\exists q' | q \xrightarrow{a} q'$
- $q \overset{\epsilon}{\Rightarrow} q'$ means $\exists q_0 = q, q_1, ..., q_n = q', \forall i \in [0, n-1], \exists \tau_i \in \{\tau\}$ such that $q_i \xrightarrow{\tau_i} q_{i+1}$
- for $a \in \Sigma^S, q \overset{a}{\Rightarrow} q'$ means $\exists q_1, q_2$ such that $q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$.
- for $a_1, ...a_n \in \Sigma^S, q \overset{a_1 ... a_n}{\Rightarrow} q'$ means $\exists q_0, ..., q_n$ such that $q = q_0 \overset{a_1}{\Rightarrow} q_1 ... \overset{a_n}{\Rightarrow} q_n = q'$.
- $\Gamma^S(q) = \{a \in \Sigma^S | q \xrightarrow{a}\}$ is the set of fir-able actions, and $in^S(q)$ (resp. $out^S(q)) = \{a \in \Sigma_i^S$ (resp. Σ_o^S) $| q \xrightarrow{a}\}$ the subsets for inputs and outputs.
- q after $\sigma = \{q' \in Q^S | q \overset{\sigma}{\Rightarrow} q'\}$

We also denote $\mathcal{A}^S$ the set of all possible actions divided into two distinct subsets $\mathcal{A}_i^S$ and $\mathcal{A}_o^S$, such that $\Sigma_i^S \subseteq \mathcal{A}_i^S$ and $\Sigma_o^S \subseteq \mathcal{A}_o^S$. All these sets are finite. The interest of the increased sets $\mathcal{A}_i^S$ and $\mathcal{A}_o^S$ is to consider events that are not handled in the original specification. It may be inputs (the tester has to choose an unexpected event), or outputs (the tester has to handle unexpected events). We consider that such sets are implicitly existing for any IOLTS.

In figure 1 we give an example of a simple IOLTS specification describing the behaviour of a phone application. The user has to pick up the phone, and to compose a number. In this situation it is possible to wait for somebody to answer (!ring), to have a signal for line occupation (!occupied), or to get informed that the number does not exist (!unknown). After the ringing signal, it is possible to begin a dialog session (!dialog). This example will be used as a specification, denoted S in the following in order to illustrate our robustness testing method.

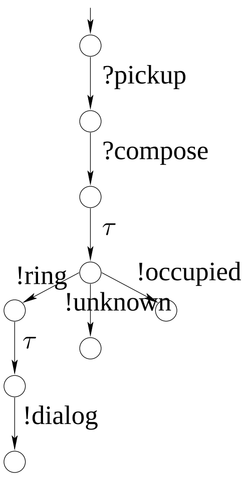


Figure 1. Example of IOLTS specification S

3 Testing method

In this section we explain our testing method. As we said before, robustness testing has to focus on unspecified parts on the specification. We use the IOLTS model to describe the specification. Thus we propose a generation method using a ioco-based approach ([TRE96]) which is sound and exhaustive. This approach gives the possibility to detect unspecified blocking states (*quiescent states*) in the implementation.

We have identified different situations that should be focused for robustness testing. We intend to check the correctness of the implementation when : (1) an input event arrives with a known symbol, but not expected at this state (2) an input arrives with an unknown symbol (3) the tester also has to be prepared to receive unexpected outputs from the system (usually leading to a fail verdict).

In order to be able to focus on unspecified parts, we have defined two new sets to extend the domain of inputs and outputs $\mathcal{A}_i^S$ and $\mathcal{A}_o^S$. The idea behind is that the tester needs to give unexpected (and unspecified) inputs sometimes. We consider that these sets are finite (which may be seen as a strong restriction, but necessary for the testing process), and practically they have to be given by the designer of the system (using experimental data for example).

This framework is divided into different steps that will be detailed in the following : (1) Construct the suspension automaton S^δ (2) Transformation the suspension automaton into a deterministic one $det(S^\delta)$ (3) Complete the obtained automaton with inputs and outputs $compl(det(S^\delta))$ (4) Select test cases for robustness.

3.1 Suspension automaton

One feature of the ioco approach is to detect quiescent (or blocking) states. So we need to identify such states in the specification, called *quiescent states*. Usually we consider that a blocking state of the implementation (practically a timeout) is considered as correct if the specification expects it.

Formally, a state q is considered quiescent in case of : (1) outputlock : $\forall a \in \Sigma_o \bigcup \{\tau\}, q \overset{a}{\not\to}$ (2) deadlock : $\forall a \in \Sigma, q \overset{a}{\not\to}$ (3) livelock : $\exists \sigma \in \{\tau\}^+, q \xrightarrow{\sigma} q$.

The suspension IOLTS S^δ is obtained from the IOLTS S by simply adding a loop labeled with a special δ output on the transition for each quiescent state of S. For a state q, we define the set $out_\delta(q) = \{a \in \Sigma_o \bigcup \{\delta\} | q \xrightarrow{a}\}$. The suspension automaton of figure 1 is given in figure 2.

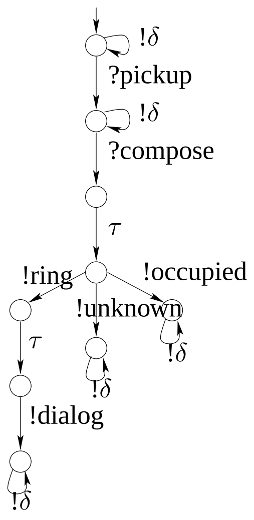


Figure 2. Suspension automaton of the specification S

3.2 Determinisation

Since we are studying black box testing, we only focus on observable actions. Moreover we need a deterministic specification in order to construct a deterministic test case. As described in [Con08], we construct a deterministic IOTLS $det(S)$ from S :

Definition 3.1 (determinisation) *$det(S) =< 2^Q, q_0$ after $\epsilon, \Sigma, \to_{det}>$ is the deterministic IOLTS obtained from $S =< Q, q_0, \Sigma, \to>$ with $\to_{det}$ the smallest relation such that for $R, R' \in 2^Q, a \in \Sigma, R \xrightarrow{a}_{det} R'$ if $R' = R$ after a.*

The deterministic IOLTS obtained from the suspension IOLTS of figure 2 is given in figure 3.

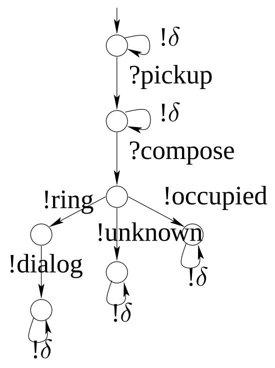


Figure 3. Determinisation of the suspension automaton

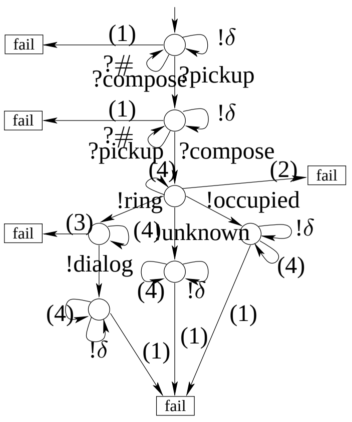
3.3 Input and output completion

This step only deals with robustness testing. We intend to complete the specification with all possible inputs and all possible outputs, and to detect the unauthorized situations leading to a fail state. Compared to conformance testing, three new aspects have to be considered : (1) We need to complete each state with all inputs known by the specification. (2) We need to complete each state with all inputs not known by the specification (increased sets). (3) We need to complete each state with all the ouputs known and and not known by the specification. This leads to a failing situation. At the end of this step, we obtain an IOLTS describing all possible behaviors for a test case.

Definition 3.2 (Completion) *Considering a deterministic suspension IOLTS $M =< Q^M, q_0^M, \Sigma^M, \to_M>$, the completion of M is a deterministic IOLTS $compl(M) =< Q, q_0, \Sigma, \to>$ such that :*

- $Q = Q_M \bigcup \{ \textit{fail} \}$
- $q_0 = q_0^M$
- $\Sigma = \mathcal{A}^M$
- $\to\ = \ \to_M \bigcup \{q \xrightarrow{a} \textit{fail} \,|a \in \mathcal{A}_o \cup \{\delta\}, q \overset{a}{\not\to}_M\} \bigcup \{q \xrightarrow{b} q | b \in \mathcal{A}_i, q \overset{b}{\not\to}_M\}$

The IOLTS in figure 4 is the result of the completion of the deterministic suspension IOLTS of figure 3. In order to simplify the notations, we use the symbol $?\#$ (resp. $!*$) to represent any symbol outside from the original input (resp output) alphabet, i.e. any element of $\mathcal{A}_i - \Sigma_i$ (resp. any element of $\mathcal{A}_o - \Sigma_o$). In a real test experimentation, $?\#$ would be replaced by any perturbing information, e.g. a corrupted message, or any information not in the specification of the system. $!*$ symbolizes any unknown event.



(1) !dialog, !ring, !unknown, !occupied, !$*$
(2) !dialog, !δ, !$*$
(3) !ring, !unknown, !occupied, !δ, !$*$
(4) ?pickup, ?compose, ?#

Figure 4. Input and output completion of the specification S

3.4 Test cases generation

At this step, we have an IOLTS $compl(det(S^\delta))$ representing all possible test cases, and especially detecting traces leading to a failing situation. This model includes robustness situations, since all possible inputs have been added, and all possible outputs are handled in the testing process. Naturally, in order to ensure completeness of generated test cases, it is necessary to consider that the sets of possible inputs ($\mathcal{A}_i$) and ouputs ($\mathcal{A}_o$) are finite.

The weakness of this IOLTS is that no particular behaviour is highlighted. For practical reasons, it is usually necessary to focus on particular behaviours. A first classic possibility is to use a test purpose in order to generate test cases. In the domain of robustness testing, it is possible to use such an approach.

In the following, we propose a test case generation focusing on robustness aspects. The idea is to force the tester to fire transitions of unexpected input situations. As we said before, we identify two possibilities of inputs hazards : (1) Apply an input known by the system, but not expected at this state (2) Apply an input that is unknown by the system.

More precisely, the user has to provide a robustness criterion composed by two integers rob_1 and rob_2. rob_1 (resp. rob_2) gives the number of successive inputs the test case has to compute in situation (1) (resp. (2)) when it is possible. It is not possible to force the system to try all unexpected input transitions since some traces depend on the answers of the implementation, i.e. some transitions may be unreachable. In the following, we consider the case (1) as an *inopportune input*, and case (2) as an *unknown input*.

Our test case generation method is given in Algorithm 2. In order to generate test cases, this algorithm has to be called on $det(S^\delta)$. Inputs and ouputs are permuted in order to adopt the tester point of view. For an input (resp. output) element a of the alphabet we use the overline notation $\overline{a}$ to express the corresponding output (resp. input) of the tester.

Algorithm 2 Test suite derivation algorithm for robustness

```
function $TC(S : IOLTS)$
{Initialisation}
**for all** state $s$ of $S$ **do**
    $s.rob1 := false$; $s.rob2 := false$
    $deriv(initial\_state(S))$
**end for**

function $deriv(s : state)$
**if** $s.rob1 = false$ **then**
    $s.rob1 := true$
    $t := \overline{a_1}; ...; \overline{a_{rob1}}; t'$ with $(a_1, ..., a_{rob1}) \in (\Sigma_i - in(s))^{rob1}$ and $t' = deriv(s$ after $a_1...a_{rob1})$
**else if** $s.rob2 = false$ **then**
    $s.rob2 := true$
    $t := \overline{a_1}; ...; \overline{a_{rob2}}; t'$ with $(a_1, ..., a_{rob1}) \in (\mathcal{A}_i - \Sigma_i)^{rob2}$ and $t' = deriv(s$ after $a_1...a_{rob2})$
**else**
    Apply one of the three following choices :
    (1) $t := pass$ {stops the recursion}
    (2) $t := \overline{a}; t'$ with $a \in in(s) \bigcup \mathcal{A}_i$ and $t' = deriv(s$ after $a)$
    {Note that $in(s) \subseteq \mathcal{A}_i$. We need to allow any element of $\mathcal{A}_i$ to ensure exhaustiveness, but practically $in(s)$ should be preferred}
    (3) $t := \Sigma\{\overline{x};$ **fail** $| x \in \mathcal{A}_o, x \notin out(s)\}$
    $+\Sigma\{\overline{\delta};$ **fail** $| \delta \notin out_\delta(s)\}$
    $+\Sigma\{\overline{x}; t_x | x \in \Sigma_o, x \in out(s)\}$ with $t_x = deriv(s$ after $x)$
    $+\Sigma\{\overline{\delta}; t_\delta | \delta \in out_\delta(s)\}$ with $t_\delta = deriv(s$ after $\delta)$
    {here $+$ and $\Sigma$ classically mean the choice and the sum}
**end if**
```

This algorithm is divided into two distinct parts. The second one is a classical non deterministic generation choice between sending an input, waiting for an output, or ending the algorithm. This part permits to get an exhaustive generation algorithm while integrating unexpected inputs in the generation. First part focuses on robustness aspects. The user has to provide two integer values, rob_1 and rob_2 which are the number of unexpected inputs we intend to inject successively (if possible) in a state : rob_1 is the number of inopportune inputs, and rob_2 is the number of unknown inputs. If possible, algorithm forces to fire rob_1 times the looping transition with inopportune inputs on this state, then rob_2

times the looping transition with unknown inputs.

A possible test case for our specification example in figure 3 is given in figure 5 with the values $(1, 1)$.

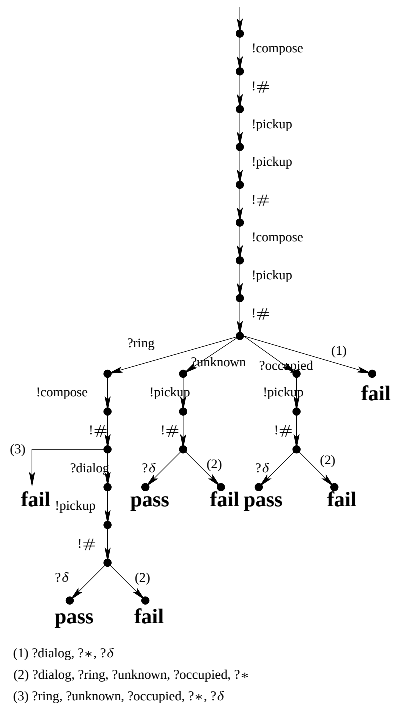


Figure 5. A possible test case for S

4 Discussion

In a summarized way, we recall that an implementation **passes** a test case if all possible executions lead to a **pass** verdict. We recall some classic properties of test cases ([TRE96]). For an implementation i and a test suite (i.e. a set of test cases) T :

- T is complete $= \forall i : i$ **ioco** $s \Leftrightarrow i$ **passes** T
- T is sound $= \forall i : i$ **ioco** $s \Rightarrow i$ **passes** T
- T is exhaustive $= \forall i : i$ **ioco** $s \Leftarrow i$ **passes** T

The soundness is the most important property of testing methods, since it is not acceptable to consider as faulty a correct implementation.

The algorithm 2 permits to generate an exhaustive test suite. In experimentation the generated test suite is sound, but of course not exhaustive. However the set of all possible test cases generated using this algorithm is exhaustive. The principle of the proof is similar to [TRE96], by considering it directly on the completed deterministic suspension automaton, using the extended alphabet $\mathcal{A}$, which is considered here as finite (exhaustiveness is possible only if the set of all possible inputs and outputs in considered as finite). Notice that second part of the algorithm (the non deterministic one) is sufficient to ensure exhaustiveness (as in [TRE96]). The first part of the algorithm permits to apply hazards in order to focus on robustness.

5 Related work

Many research have been done in the domain of reactive systems testing . The majority of these works deals with conformance testing, normalized in [IEE04]. An overview may be found in [JER03]. In this section, we focus particularly on robustness testing works.

In [CW03], authors propose 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 [RLT02], 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 [BCSS90] modifies a processus binary image in memory. In [Reg05], authors propose to apply randomly interruptions in the IUT, whereas the BALLISTA tool works on data unexpected modifications. This idea is explained in [DKG99]. 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 [SKRC07, FMP05, Rol03].

In [SKRC07], authors propose an approach based on an increased specification used to specify the acceptable behaviours in presence of hazards.

In [FMP05], 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 [Rol03] 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 [Rol03] and this work is in the concept of robustness : we consider here that robustness implies conformance; the method described in [Rol03] does not.

In our approach, we use only the original specification, but the extended alphabet has to be provided.

6 Conclusion and future works

In this paper, we have presented a formal approach to test the robustness of a system modeled with an IOLTS. In this work, we have considered an increased alphabet of the system, and we have proposed a way to inject these hazards in the test cases. We have proposed a test case generation algorithm permitting to obtain a sound and exhaustive test suite.

In the future, we intend to extend this approach on real time systems. The idea is to model the specification using timed automata with inputs and outputs. The difficulty of this work is to handle new kinds of hazards due to timing failures. In a general way, timing aspects may lead to combinatorial explosion. In order to solve this problem, we intend to use the so called difference Bounds Matrix (DBM).

References

[BCSS90] 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.

[Con08] C. Constant. *Génération automatique de tests pour modèles avec variables ou récursivité*. PhD thesis, Université de Rennes 1, 2008.

[CW03] R. CASTANET and H. WAESELYNK. Techniques avances de test de systmes complexes: Test de robustesse. Technical report, Action spcifique 23 du CNRS, 11 2003.

[DKG99] J. DeVale, P. Koopman, and D. Guttendorf. The ballista software robustness testing service. In *Testing Computer Software Conference (TCSC99)*, June 1999.

[FMP05] 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.

[FTdV06] Lars Frantzen, Jan Tretmans, and René de Vries. Towards model-based testing of web services. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Sicily, ITALY, June 9th 2006.

[IEE04] IEEE. *International Organization for Standardization, Conformance testing methodology and framework - part 2: abstract test suite specification*, 2004.

[JER03] T. JERON. Gnration de tests pour les systmes ractifs. un survol des thories et techniques. In IRIT, editor, *ETR2003. Systmes, Rseaux et Applications*, pages 105–122. IRIT, Septembre 2003.

[oSET99] IEEE Standard Glossary of Software Engineering Terminology 610.12-1990. Customer and terminology standards. *In IEEE Standards Software Engineering, IEEE Press*, 1, 1999.

[Reg05] 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.

[RLT02] J. Rning, M. Laakso, and A. Takanen. PROTOS - systematic approach to eliminate software vulnerabilities. http://www.ee.oulu.fi/research/ouspg, May 2002. 2002.

[Rol03] 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.

[SKRC07] F. Saad-Khorchef, A. Rollet, and R. Castanet. A framework and a tool for robustness testing of communicationg software. In *22nd annual ACM Symposium on Applied Computing (SAC'07), March 11-15, 2007 Seoul, Korea*, pages 1461–1466. ACM Press, march 2007.

[TRE96] 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.