Two complementary approaches to test robustness of reactive systems

Antoine Rollet
LABRI - CNRS (UMR 5800)
University of Bordeaux 1
33405 Talence cedex, France
rollet@labri.fr

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

Abstract

Robustness is an important aspect for reactive embedded systems. In this paper, we present two complementary approaches dedicated to test the robustness of reactive systems modeled in the IOLTS model. The first approach uses the specification to build a hazards matrix permitting to generate cases. The other one is based on two specifications, a nominal and a degraded one describing the minimal required behaviour of the system.
Key-words *: Robustness Testing, Labelled Transition Systems, Hazards, Robustness relations.*

1 Introduction

Embedded systems are usually exposed to stressful conditions in situations where a failure often leads to catastrophic consequences. Thus it is really important to test properly conformance and robustness of such systems in order to increase the security. Since systems are getting complex, the use of automatic testing methods is necessary to obtain a sufficient level of confidence.

The IEEE defined the robustness notion as: "a system is considered as robust if it is able to operate correctly in the presence of invalid inputs or stressful environment". Then we intend to evaluate the behaviour of the system in case of unexpected situations, called hazards. We present two approaches; in both, specifications are written in the IOLTS model ([Tre96]): (1) An approach consisting in using the specification of the system and the unexpected events to construct an increased specification, called "hazards matrix". (2) Another approach consisting in using two specifications, a nominal one, describing the system in nominal conditions, and a degraded one, giving the minimal required behaviour in case of critical situation. We consider it as the "vital behaviour".

The two methods are complementary. The first one can be used if the designer considers that robustness implies robustness. The second one assumes that in case of critical situation, conformance may be lost, but the system must respect a minimal behaviour. In this case, the verdict is given using the degraded specification as a reference. Note that in both cases, we are working with a black box testing approach.

This paper is structured as follows. Section 2 contains related works on robustness testing. In section 3, we give some explanations on hazards and define some models. Section 4 presents the robustness testing method with the increased specification approach, and section 5 gives the describes the approach based on hazards injection. Section 6 gives the conclusion and some ideas about future works.

2 Related work

Many studies have been focusing on testing reactive systems. A major part of them are dedicated to conformance testing. However, robustness testing has been studied in some areas. In the following, we give an overview on robustness testing and fault injection works.

Some studies are based on fault injection by using special-purpose hardware which causes electric disturbances.

[KFA$^+$95] proposes in the MARS architecture a comparison of three kinds of physical fault injection: heavy-ion, pin-level and ElectroMagnetics. These approaches can be classified as destructive testing techniques. They may cause permanent damages on systems, then they are really costly.

Some other fault injection approaches applied to software are developed and are denoted Software Implemented Fault Injection (SWIFI). Among, these studies, the FIAT system ([BCSS90]) modifies the binary image of a process in memory (fault injection at compile time).

These approaches have lower cost than the hardware one, and it permits to specify which fault to inject, and the number of times it has to be injected. In the sofware oriented approach, [Reg05] proposes to apply randomly unexpected interrupts on the system.

A research team from Carnegie Mellon University proposed an object oriented approach to test software robustness based on parameter data types rather than component functionality with a blackbox approach. It is implemented in the Ballista tool ([DKPD99]).

Ballista is based on automatic creation and execution of invalid input robustness tests.

[AS202], a study on formal robustness testing for embedded systems, suggests mainly to consider all possible faults that a system can execute. It gives a state of the art of robustness testing and the different ways to handle the problem.

In [JCFP05] and [SKD05], authors model all possible faults directly in the specification, building a "mutant" or "degraded" specification. The system and the hazards are modeled as Input Output Labelled Transition Systems. Finally, the "mutant" specification is used to generate test sequences applied on the IUT. In [SKD05], the $ioco$ conformance relation is used to give a verdict of robustness.

3 Hazards and models

A hazard can be defined as "any event not expected by the system". By extension, we consider as hazard any event not expected in the specification of the system (meaning that the choice of the model may change the possibilities of hazards). In [AS202], three kinds of hazards are considered : (1) internal hazards : an unexpected event inside the system (e.g. a physical problem); (2) external hazards : an unexpected event on the fronteers on the system (e.g. a faulty input); (3) out of fronteers hazards : an unexpected event (far) away, but having consequences on the system (e.g. radiations, heat, ...). In our method, only external hazards are considered, since it is the only category which is controllable.

In both methods, the specifications are described in the IOLTS model ([Tre96]) :

Definition 3.1 (IOLTS) *An Input Output Labelled Transition System (IOLTS) is a 4-uplet* $< Q, q_0, \Sigma, \rightarrow >$ *such that :*

- Q *is a countable set of states;*
- $q_0 \in Q$ *is the initial state;*
- Σ *is a countable alphabet of actions, partitioned into two subsets such that* $\Sigma = \Sigma_i \cup \Sigma_o$ *(respectively input marked with a* ? *and ouputs marked with a* !*);*
- $\rightarrow \subset Q \times \Sigma_\tau \times Q$ *is a set of transitions (*Σ_τ *means* $\Sigma \cup \{\tau\}$ *where* τ *is an internal action).*

We also denote $\mathcal{A}$ the set of all possible actions divided into two distinct subsets $\mathcal{A}_i$ and $\mathcal{A}_o$, such that $\Sigma_i \subseteq \mathcal{A}_i$ and $\Sigma_o \subseteq \mathcal{A}_o$.

4 Increased specification method

As we discussed before, this method is developed with the idea that a robust system must also conform to the specification. Thus, the principle here is to identify carefully controllable and observable hazards for the system, and to use them in order to generate test cases. More precisely, we consider two different actors. Firstly, the "tester" uses the specification modeled as an IOLTS ([Tre96]) to identify the hazards of the system. Then, the default behaviour is automatically computed. This increased behaviour is given to the "designer" who can decide to precise some specific behaviours. Finally, this increased specification is used by the "tester" in order to generate complete test cases.

In order to generate test cases, six steps are necessary; the specification is given is the IOLTS model :

1. The specification is stored in an array considering controllable and observable hazards of the alphabet. We denote it Hazards matrix (H).
2. Behaviour in case of timeout (quiescent states) is computed in H
3. Behaviour of non specified inputs is computed in H
4. Behaviour of non specified outputs is computed in H
5. The special behaviour for particular hazards is computed in H
6. Test cases are generated using H

4.1 Detailed framework

In this part, we describe precisely each step of the method. In the same time, we consider a simple example in order to illustrate each step. The system is described in the IOLTS $S = < Q, q_0, \Sigma, \rightarrow >$.

In the following we consider as a usual test hypothesis that the implementation may be modeled as an IOLTS and that the system is deterministic. Furthermore, we do not handle τ transitions for a simplicity reason and because they are not observable and then not necessary for testing. In case of τ transition of the specification, it is easy to remove them (using the $after$ operator, see [Tre96]).

4.1.1 Step 1 : computing the hazards matrix (H)

For each state we have to consider all possible events. In this matrix, lines are labeled with state names, and columns are labeled with all possible actions of $\mathcal{A}$. Because, we do not know all the actions (we only know Σ), in fact we create one column for each action of Σ, and we add a column for any other input, and another one for any other output. We

also need to add a column to specify the behaviour in case of quiescent state, which is a "special action" (see next step).

Consider for example the specification S of figure 1.

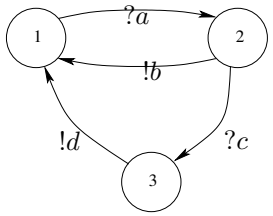

Figure 1. Example of IOLTS specification

Then we obtain the following hazards matrix :

	$?a$	$?c$	$\mathcal{A}_i - \{?a, ?c\}$	$!b$	$!d$	$!\delta$	$\mathcal{A}_o - \{!b, !d\}$
1	2						
2		3		1			
3					1		

The rule of the following steps is to fill the empty cells and eventually to modify some of them. However it is important to notice that the cells filled in this first step can not be modified later (so that conformance of the system is kept).

4.1.2 Step 2 : computing the behaviour in case of time-out

The absence of response may be something described in the specification. In [Tre96], authors define special states in the specification called quiescent states. These states can not evolve without an external input. Such situation is considered as observable with a special output not in $\mathcal{A}$ written $!\delta$. In practice, the tester identifies a timeout. We adapt the definition of quiescent state of [Tre96] using directly the hazards matrix : a state is quiescent if, considering the corresponding line in the matrix, all crossing columns labeled with an output (i.e. any element of $\mathcal{A}_o$) are not filled. The default behaviour in case of quiescent output is : (1) if the source state is quiescent, the system has to loop back in the same state (2) otherwise it is a $fail$ situation In the following, we consider the notation : for a state S of Q and an action a in $\mathcal{A}$, we note $H(S, a)$ the cell of H intersection of the line labeled by S and the column labeled by a if the column exists, otherwise the column labeled by the set containing a. The $fail$ verdict will be written f in the matrix.

Thus, the matrix is filled as follows (H is the hazards matrix):

for all state $S_i \in Q$ **do**
  if quiescent(S_i) **then**
    $H(S_i, !\delta) := S_i$
  else
    $H(S_i, !\delta) := fail$
  end if
end for

In our example, we obtain the following matrix H :

	$?a$	$?c$	$\mathcal{A}_i - \{?a, ?c\}$	$!b$	$!d$	$!\delta$	$\mathcal{A}_o - \{!b, !d\}$
1	2					**1**	
2		3		1		**f**	
3					1	**f**	

4.1.3 Step 3 : computing the behaviour in case of unexpected input

In this step, we identify and compute the default behaviour in case of unexpected input. Usually, systems should accept the input and stay in the same state. Then, we consider that the system has to loop back in the system state in such a situation. In our work, this gives :

for all state $S_i \in Q$ **do**
  for all label $a \in \mathcal{A}_i$ **do** {only inputs are considered}
    if $H(S_i, a) = \emptyset$ **then**
      $H(S_i, a) := S_i$
    end if
  end for
end for

In our example, this gives :

	$?a$	$?c$	$\mathcal{A}_i - \{?a, ?c\}$	$!b$	$!d$	$!\delta$	$\mathcal{A}_o - \{!b, !d\}$
1	2	**1**	**1**			1	
2	**2**	3	**2**	1		f	
3	**3**	**3**	**3**		1	f	

4.1.4 Step 4 : computing the behaviour in case of unexpected output

This step is similar to the previous one. We finish to fill the matrix considering the outputs not expected in a particular situation. The default behaviour in such a situation is to assume that any unexpected output leads to a $fail$ situation. This gives :

for all state $S_i \in Q$ **do**
  for all label $a \in \mathcal{A}_o$ **do** {only outputs are considered}
    if $H(S_i, a) = \emptyset$ **then**
      $H(S_i, a) := fail$
    end if
  end for
end for

And we obtain the matrix :

	$?a$	$?c$	$\mathcal{A}_i - \{?a, ?c\}$	$!b$	$!d$	$!\delta$	$\mathcal{A}_o - \{!b, !d\}$
1	2	1	1	**f**	**f**	1	**f**
2	2	3	2	1	**f**	f	**f**
3	3	3	3	**f**	1	f	**f**

4.1.5 Step 5 : handling the hazards

A this moment, the possible hazards are identified and the default behaviour is computed in the matrix. This matrix is provided to the designer of the system who has the possibility to modify some cells. It is important to notice that it is forbidden to modify any cell filled in the first step, so that the system keeps conformance with the original specification. For example, if we consider the specification of the TCP protocol ([]), it does not consider the reception of a RST input in any state. However, usual implementations assume that a RST message leads to the initial state. Sometimes, some outputs may be acceptable, with a loopback on the same state... In our example, it is possible to imagine that in case of $?a$ input occurring in state 2, the system comes back to the first state (a kind of reset). This would give the matrix :

	$?a$	$?c$	$\mathcal{A}_i - \{?a, ?c\}$	$!b$	$!d$	$!\delta$	$\mathcal{A}_o - \{!b, !d\}$
1	2	1	1	f	f	1	f
2	**1**	3	2	1	f	f	f
3	3	3	3	f	1	f	f

Remark that it could be reasonable not to allow the modifications of the last column (labeled with the set of unexpected outputs)

This transformation is simple and light. However, we have at this step all possible representable behaviours. More precisely we have in this matrix all possible inputs and the verdicts. It is now easy to use it to generate a complete test suite.

Using the same idea, it would be possible to allow the designer to add some columns in the matrix, in case for example of particular behaviour for a precise hazard. The principle is completely similar : we just have to add a column labeled with this input (resp. output), and to remove it from the column of all other possible inputs (resp. outputs).

4.1.6 Step 6 : Test derivation

In this part, we use the hazards matrix in order to generate test cases. The originality here is that the verdicts are directly included in the matrix. This simple algorithm 1 can produce a complete test suite T for the $ioco$ relation (see [Tre96]). The proof of completeness is similar to [Tre96] since the hazards matrix can be seen as a canonical tester (in fact the mirror image, i.e. inputs (resp. outputs) get outputs (resp. inputs)). In order to generate a test case, we consider a recursive function $deriv$ building a tree. At the beginning of the generation, this function is applied on the initial state of the specification, i.e. $deriv(q_0)$. For an action a, we use the notation $\bar{a}$ to express the tester point of view, i.e. an input (resp. output) gets an output (resp. input).

In algorithm 1, we have three possibilities. We can stop the derivation, send an input (always possible because of the hazards), or wait for an answer from the system. We remark that in the second rule, if the choice is made randomly and with equiprobability on the set of inputs $\mathcal{A}_i$, this implies that the test case focuses on hazards (unexpected inputs), since the set of unexpected inputs in a given state is usually important compared to the set of expected ones.

Algorithm 1 Test suite derivation algorithm

The test case is obtained from H by a finite number of recursive applications of one of the following three choices :

function $deriv(s : state)$ ::=
  if $s \neq fail$ **then**
    (1) $t := pass$ {stops the recursion}
    (2) $t := \bar{a}; t'$ where $a \in \mathcal{A}_i$ with $t' := deriv(H(s, a))$ {we (randomly) choose one input $a \in \mathcal{A}_i$}
    (3) $t := \sum\{\bar{x}; t_x | x \in \mathcal{A}_o \bigcup \{\delta\}\}$ with $t_x := deriv(H(s, x))$ {$\sum$ stands for the sum of all expressions in a set}
  else
    $t := fail$
  end if

Figure 2 illustrates an example of test case obtained with our example using the following rules sequence : 2 - 3 - 2 - 2 - 2 - 3 - 1. We use the notation $?*$ for all other actions received by the tester, and $!*$ for all other actions sent by the tester. In practice for the last case, the tester chooses an action to send in the extended alphabet $\mathcal{A}_i - \Sigma_i$.

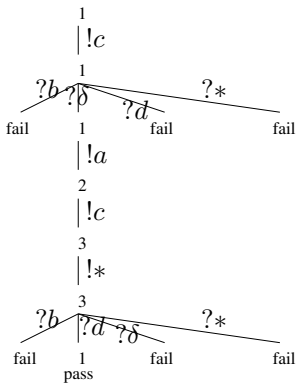

Figure 2. Example of test derivation

The obtained test has the property to be complete (i.e. sound and exhaustive) with the $ioco$ relation. The proof is similar to [Tre96] with the (obvious) idea that for a state q and an action a, $s = H(q, a) \Rightarrow s = q\ after\ a$.

5 Hazards injection method

Now we consider in this section that robustness does not imply conformance. The idea here is that in case of unexpected conditions, the system gets into a degraded mode. Only the vital functionalities are required. Notice that in this approach, we do not know precisely "where" the real behaviour is between the two modes, and the environment does not always know in which mode the system is (see FIG. 3).

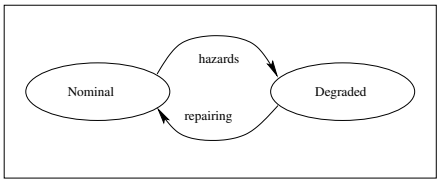

Figure 3. Nominal and degraded behaviour

An example of such reasoning is the domain of embedded calculators in aircrafts. In nominal mode, the calculators are used for the safety of passengers, but also for comfort, whereas in case of unexpected situation (implying degraded mode), the aim of the calculators is to lead the aircraft to an airport, without taking care of comfort. Such ideas are currently used in aeronautics, but generally with two different systems for the nominal and the degraded mode.In our work, the system is described with two specifications modeled as IOLTS, a *nominal* one, S, describing the behaviour of the system in normal conditions, and a *degraded* one, S', describing the behaviour in critical situation, i.e. giving the vital functionalities and the minimum required behaviour, particularly in the case of unexpected situations. In the following, if no precision is given, Σ means the set of labels of S and Σ' means the set of labels of S'.

The methodology is composed of four steps :

1. test sequences are derived using the nominal specification
2. Hazards are injected in these sequences; **only non vital actions can be modified**, i.e. only actions which are not member of the alphabet of S'
3. test sequences are applied, traces recorded
4. traces are check to give a robustness verdict

We consider that the two specifications of the system can not be completely independant. There is a kind of "behavioural" inclusion of the behaviour between S and S', since the vital behaviour of a system has to be included in the normal one. Then, there is a relation $S \leq_{rob} S'$ (described below) for the IOLTS models, modeling the fact that for all trace in S', we can find a trace of S having the same actions in the same order. FIG. 4 shows an example of the relation $S \leq_{rob} S'$ defined in Def. 5.2 : in this figure, the relation $S \leq_{rob} S'$ is true. The generated test sequences contain events that may not be included in the degraded specification, such events are identified as *non-vital* events, other events are identified as *vital* events. In our work, the set of vital events or actions is the alphabet of the degraded specification (Σ').

The system is specified with a nominal specification $S = < Q, q_0, \Sigma, \rightarrow >$ and a degraded specification $S' = < Q', q'_0, \Sigma', \rightarrow' >$ such that $S \leq_{rob} S'$ (implying that $\Sigma' \subset \Sigma$). This relation is defined just below (Def 5.2). The set of **vital actions** is defined as the alphabet of the degraded specification, here Σ'.

5.1 Definitions

In this part, we give some definitions needed to describe our method.

Definition 5.1 (ordered projection $\Pi_{\Sigma'}$) *We define recursively the ordered projection function $\Pi_{\Sigma'}$:*

$$\begin{array}{lll} \Sigma^* & \rightarrow & (\Sigma')^* \\ \sigma = (a_1 ... a_n) & \rightarrow & \begin{cases} \emptyset \text{ if } \sigma = \emptyset \\ a_1.\Pi_{\Sigma'}(a_2...a_n) \text{ if } a_1 \in \Sigma' \text{ and } \sigma \neq \emptyset \\ \Pi_{\Sigma'}(a_2...a_n) \text{ else } (\sigma \neq \emptyset \text{ and } a_1 \notin \Sigma') \end{cases} \end{array}$$

where "dot" means the concatenation.

In other words, $\Pi_{\Sigma'}(\sigma)$ is the subsequence of $\sigma = (a_1...a_n)$ containing only the elements of Σ' (with the same order and repetitions). Suppose for example that the set $\Sigma = \{?a, ?b, !c, !d, !e, !f\}$ and $\Sigma' = \{?a, !c, !e\}$, then if we apply this function on a trace $\sigma = ?a.?b.!c.!d$, we obtain $\Pi_{\Sigma'}(?a.?b.!c.!d) = ?a.!c$.

Now it is possible to define the relation $\leq_{rob}$ between S and S'.

Definition 5.2 ($\leq_{rob}$) *Let two IOLTS S and S', we say that S' is included in S in the sense of robustness iff : $S \leq_{rob} S'$ $=_{def} \forall \sigma' \in Traces(S'), \exists \sigma \in Traces(S)$ such that $\sigma' = \Pi_{\Sigma_{S'}}(\sigma)$*

As we said, the aim of $\leq_{rob}$ is to ensure a relation between the nominal and the degraded mode. We consider in fact that all possible behaviours (i.e. traces in our case) must be described in the nominal specification. The idea is that for any trace σ of S' (in other words a trace of vital actions), there must exist a trace in S "including" the behaviour of σ (i.e. having all the vital actions in the correct order).

For example, if we consider FIG. 4, we have :

- $S \leq_{rob} S'$ since there exists for each trace σ of S' a trace of S which is the ordered projection of σ over

the alphabet of S'. Indeed, the trace $?x!y?a!c$ of S "includes" the trace $?a!c$ of S', and the trace $?b!g!d$ of S "includes" the trace $?b!d$ of S';

- $I_1 \leq_{rob} S'$ since the trace $?x!y?a!y!c$ of I_1 "includes" the trace $?a!c$ of S', and the trace $?b!d$ of I_1 "includes" the trace $?b!d$ of S';

- $\neg(I_2 \leq_{rob} S')$: there is no trace in I_2 "including" the trace $?b!d$ of S'.

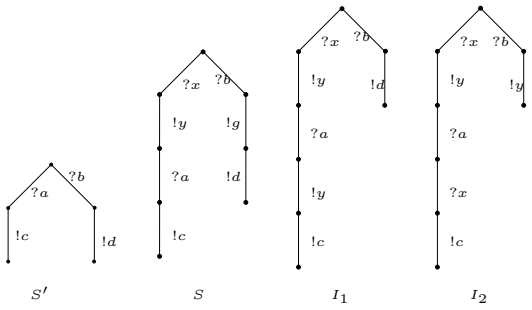

Figure 4. Examples for $\leq_{rob}$ relation

5.2 integration of hazards

As we discussed previously, only external hazards are considered, since it is the only category which is controllable. We generate the test sequences from the specification. Then, we apply an hazards integration directly in the sequences, and we apply these "mutant" sequences on the IUT. The integration rules are not the topic of this paper. More details about this integration can be found in [].

The sequences contain vital and non vital actions. Consequently, there are many possibilities of modifications on the sequences, keeping in mind that we can not modify vital actions. Different operations are possible on non vital actions. We propose :
(1) adding, deleting, permuting non vital actions; (2) failure scenario of another component communicating with the tested one; (3) another communicating component has turned into degraded mode.

Suppose for example that a generated sequence is : $?a$, $!b$, $?a$, $!b$, $?e$, $!j$, $?g$, $!i$, $?c$, $!d$ with Σ' = $\{?a, !b, ?c, !d\}$, Σ = $\{?a, !b, ?c, !d, ?e, ?f, ?g, ?h, !i, !j, !k\}$ and $?l \in \mathcal{A}$. A possible modified sequence could be : $?a$,$?\mathbf{f}$, $!b$, $?a$, $!b$, $!j$, $?g$, $!i$, $?\mathbf{l}$, $?c$, $!d$.

In fact, the sequence is valid if all the vital actions are still in the resulting sequence, and in the same order. In our case, the input actions $?f$ and $?l$ has been added, and the input action $?e$ has been removed.

5.3 Behaviour analysis

At this step we have to control if the recorded behaviour may be considered as acceptable. Here "acceptable" means that the minimal vital behaviour is ensured. As we can see in FIG. 5, we do not know in advance the actual system behaviour and its degradation degree, we only know that it has to be "between" two defined specifications : the nominal one and the degraded one. In fact, we will check in the implementation traces if the minimal behaviour described in the *degraded* specification is ensured by the IUT, but we do not take care about actions not described in the degraded specification. Then, the degraded specification will be the reference to decide if the implementation is robust. We assume in a classic that the implementation can be modeled in the IOLTS model.

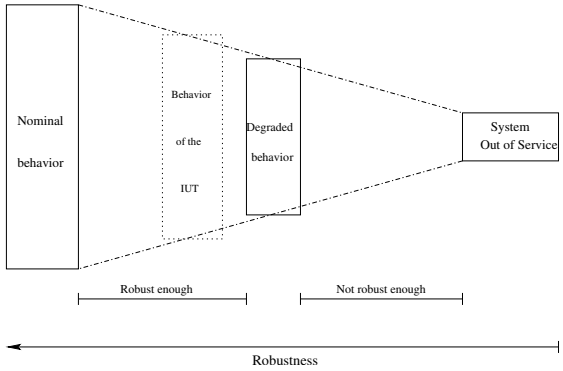

Figure 5. Robustness degree

A way to define a robust behaviour is that any trace of the degraded specification should be found in the behaviour of the implementation, i.e. the same actions in the same ordering. These vital actions could be eventually mixed with non vital actions, but all vital actions has to be present in the same ordering.

Finally we consider an implementation I as robust if in any case of hazards integration (included in I), $I \leq_{rob} S'$. If we see FIG. 4, supposing that the system is modeled by S (nominal specification) and S' (degraded specification), and that I_1 and I_2 are two obtained implementations, then we can say that I_1 is robust (because $I_1 \leq_{rob} S'$), and that I_2 is not (because $\neg(I_2 \leq_{rob} S')$) .

We also consider that the specifications S' must be included in the sense of robustness in S ($S \leq_{rob} S'$) so that an implementation I having exactly the same behaviour of S is also considered as robust (soundness of the test), and so that all sequences generated from the nominal specification S contain inevitably vital input actions in the correct order, allowing us to identify the corresponding vital ouputs.

Finally, it is possible to test the robustness with our

method if $S \leq_{rob} S'$ and the final implementation I of the system is considered robust if $I \leq_{rob} S'$.

6 Conclusion

In this paper, we have presented two complementary approaches to test the robustness. The major difference between both is the fact that robustness implies conformance or not. The first method identifies some classic hazards and uses them in order to create a hazards matrix, whereas the other one directly integrates the hazards in the test sequences.

We intend to define precisely the robustness degrees and a method able to qualify formaly the degraded degree of any implementation. We also intend to work on robustness symbolic testing of systems described by models using data.

References

[AS202] Action spécifique 23 du cnrs, département stic: Test avancé de systèmes complexes, test de robustesse, 2002. Animateurs: Richard Castanet et Hélène Waeselynck.

[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.

[DKPD99] DeVale, J. Koopman, P., and Guttendorf D. The ballista software robustness testing service. In *Testing Computer Software Conference (TCSC99)*, June 1999.

[JCFP05] Laurent Mounier Jean-Claude Fernandez and Cyril Pachon. A model-based approach for robustness testing. In Rachida Dssouli Ferhat Khendek, editor, *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005*, lncs, pages 333–348. Springer, June 2005.

[KFA+95] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber. Integration and comparison of three physical fault injection techniques. In *Predictably Dependable Computing Systems, chapter V: Fault Injection*, pages 309–329. Springer Verlag, 1995.

[Reg05] John 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.

[SKD05] F. Saad Khorchef and X. Delord. Une méthode de test de robustesse adaptée aux protocoles de communication : application au protocole tcp. In *CFIP'2005, Bordeaux, France (Papier Court)*, Mars 2005.

[Tre96] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17:103–120, 1996.