Remote testing can be as powerful as local testing*

Claude Jard, Thierry Jéron, Lénaïck Tanguy & César Viho
IRISA - Campus de Beaulieu, F35042 Rennes, France
e-mail: {jard, jeron, ltanguy, viho}@irisa.fr

This work is partially supported by the French Army (CELAR).
C. Jard is member of CNRS, T. Jéron of INRIA,
L. Tanguy and C. Viho of IFSIC/University of Rennes I

Abstract
Designing test cases for remote asynchronous testing is error-prone. This is due to the difficulty to foresee all the disorders on the observations collected by the tester as well as the possible collisions between stimuli and observations. Designing correct synchronous test cases is easier, but transforming them into correct asynchronous ones is a difficult task. Moreover, it is difficult to compare remote testing and local testing as in general sets of conformant implementations are not comparable.

In this paper, we prove that by the use of logical stamps, remote testing can gain the same power as local testing: the conformant implementations in an asynchronous environment are exactly the same ones as in a synchronous environment. We give an operational method to derive the correct test cases for remote testing with this testing power. Furthermore, we show that test cases designed for a synchronous environment can test synchronous conformance in an asynchronous environment. This is achieved by the implementation of a test driver executed at runtime on the tester.

Keywords
Conformance Testing, Test Generation, Local and Remote Testing, Asynchronism, Stamp

1 INTRODUCTION

We consider the context of black box conformance testing in which an implementation under test (IUT for short) is tested in order to obtain the conviction that its behaviour conforms with its specification. The tester stimulates the IUT by sending messages on points of control and observation (PCOs) and observes on these same PCOs the reactions of the IUT (see figure 1, part A). Within sight of the reactions, a verdict (Fail, Pass or Inconclusive) is emitted. The underlying concepts have been formalized since the last years leading to the so called testing theory which identifies the notion of formal conformance

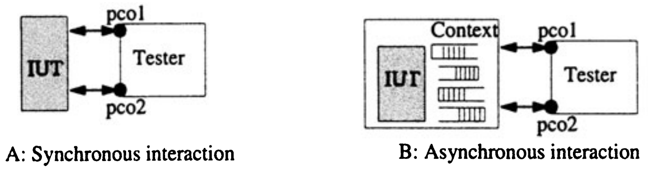

Figure 1 Synchronous and asynchronous testing

relation and gives a precise meaning of the verdicts (see [11] for example). Originally, the theory considered a synchronous interaction between the tester and the IUT. This made the implicit assumption that an IUT can refuse an event and that the tester can observe the refusal [1]. In practice however, one cannot always avoid taking into account the test environment intercalated between the tester and the IUT. As advocated in [4], conformance must then be defined as the conformance of the IUT in its test environment with the specification in a model of this environment. The most frequent example is that of remote testing architecture in which the tester reaches the IUT through a network connection. In this case, PCOs can be seen as composed of two FIFO queues, one for each direction of the interaction. The communication between the tester and the IUT is then asynchronous as illustrated in figure 1, part B.

The asynchronous nature of the PCOs poses some difficulties to design correct test cases. The possibility of disorder on the observations collected on different PCOs as well as the possible collision on a PCO between stimuli and observations have to be taken into account. This should be achieved by the inclusion of this asynchronous behaviour in the specification. But during the examination of existing test suites, one realizes (see for instance [2]) that this is not always done and it is the main reason of non-validity of some tests.

Another problem is the fundamental difference between conformance in synchronous or asynchronous environment. Precisely, for a particular specification, the sets of conformant IUTs in synchronous or asynchronous environment are in general not comparable. This explains why transforming a correct synchronous test case into a correct asynchronous test case is a difficult task.

It appears nevertheless that under certain conditions, the asynchronous deformation is foreseeable and even invertible. This topic is investigated in this paper and our main results are shortly described here:

1. We show that the **ioconf** conformance relation [12] has a simple characterization in terms of trace inclusion for input complete specifications. Moreover a specification can always be completed without modification of its set of conformant implementations.
2. We deduce that for complete specifications, synchronous conformance is preserved by the asynchronous environment. This means that a conformant IUT in synchronous environment is still conformant in an asynchronous environment. But asynchronous conformance is still more permissive: some

non conformant IUTs in synchronous environment can be conformant in an asynchronous environment.

3. Under the assumption of one PCO linking the IUT and its environment, and by using stamps (a simple counting mechanism), we prove that the permissiveness of remote asynchronous testing can be suppressed: conformant IUTs in synchronous environment are exactly the same ones as in asynchronous environment. From the preceding results, we deduce a method for the automatic generation of tests with synchronous testing power.
4. Furthermore, still in the case of one PCO, synchronous test cases can be used for asynchronous testing. This is achieved by a specific driver which inverts the asynchronous transformation using stamps. We think that this new mechanism can be of a great utility. It makes possible to conceive tests in a synchronous way while carrying out them in an asynchronous environment.

The presentation is organized as follows. We start by the definition of models. Then, we recall the local synchronous approach and the need to complete specifications to prepare their use in an asynchronous environment. Secondly, we present the distortions induced by the PCO queues and deduce the asynchronous conformance relation. Then we define the counting mechanism in the case of one PCO. We deduce a transformation of the specification allowing to generate automatically test cases for asynchronous testing with the power of synchronous testing. The next part is devoted to a possible implementation of a test driver which controls a synchronous test case in an asynchronous environment with the information acquired dynamically from the IUT. We finish by some prospects of generalization in the case of multiple PCOs.

2 LOCAL SYNCHRONOUS TESTING

Because of the asymmetrical nature of the testing activity, the models have to differentiate input and output actions. In this paper, we will use the model of IOLTS (Input-Output Labeled Transition Systems) to describe the different objects involved in the conformance testing.

2.1 Models

Definition 21 *An IOLTS is a tuple* $M{=}(Q^M, P_I^M, P_O^M, A_I^M, A_O^M, T^M, q_{init}^M)$ *where* Q^M *is a set of states,* $q_{init}^M \in Q^M$ *is the initial state,* P_I^M *and* P_O^M *are finite sets of input and output ports.* A_I^M *and* A_O^M *respectively are finite input and output alphabets.* $A^M \subseteq P_I^M \times \{?\} \times A_I^M \cup P_O^M \times \{!\} \times A_O^M$ *is the alphabet of observable actions constructed from the sets of input-output ports and input-output alphabets.* $\tau \notin A^M$ *denotes an internal action.* $T^M \subseteq Q^M \times A^M \cup \{\tau\} \times Q^M$ *is the transition relation, we note* $p \overset{\alpha}{\rightarrow}_M q$ *for* $(p, \alpha, q) \in T^M$.

Let $\alpha_i \in A^M$, $\mu_i \in A^M \cup \{\tau\}$, $\sigma \in (A^M)^*$, $q, q', q_i \in Q^M$:

- $q \xrightarrow{\mu_1 \cdots \mu_n} q' =_{def} \exists q_0 = q, q_1 \ldots, q_n = q', \forall i \in [1, n], q_{i-1} \overset{\mu_i}{\rightarrow} q_i,$

• $q \xrightarrow{\mu_1 \cdots \mu_n} =_{def} \exists q', q \xrightarrow{\mu_1 \cdots \mu_n} q'$ and $q \overset{\mu_1 \cdots \mu_n}{\not\longrightarrow} q' =_{def} \neg(q \xrightarrow{\mu_1 \cdots \mu_n})$,
• $q \overset{\epsilon}{\Rightarrow} q' =_{def} q = q'$ or $q \xrightarrow{\tau \cdots \tau} q'$ and $q \overset{\alpha}{\Rightarrow} q' =_{def} \exists q_1, q_2, q \overset{\epsilon}{\Rightarrow} q_1 \overset{\alpha}{\rightarrow} q_2 \overset{\epsilon}{\Rightarrow} q'$,
• $q \overset{\alpha_1 \cdots \alpha_n}{\Longrightarrow} q' =_{def} \exists q_0 = q, q_1 \ldots, q_n = q', \forall i \in [1, n], q_{i-1} \overset{\alpha_i}{\Rightarrow} q_i$,
• $enable(q) =_{def} \{\alpha \in A^{\mathrm{M}} \mid \exists q'$ and $q \overset{\alpha}{\Rightarrow}_{\mathrm{M}} q'\}$ is the set of observable actions possible in q, $In(q) =_{def} \{a \in A_I^{\mathrm{M}} \mid \exists p \in P_I^{\mathrm{M}}, p?a \in enable(q)\}$ is the set of possible inputs in q, and $Out(q) =_{def} \{a \in A_O^{\mathrm{M}} \mid \exists p \in P_O^{\mathrm{M}}, p!a \in enable(q)\}$ is the set of possible outputs in q,
• $q \ after \ \sigma =_{def} \{q \in Q^{\mathrm{M}} \mid q \overset{\sigma}{\Rightarrow}_{\mathrm{M}} q\}$ is the set of reachable states from q by the sequence of observable actions σ,
• $Traces(q) =_{def} \{\sigma \in (A^{\mathrm{M}})^* \mid q \ after \ \sigma \neq \emptyset\}$,
• if $\alpha \in A^{\mathrm{M}}$ is an observable action, we note $\overline{\alpha}$ its mirror action: if $\alpha = p!a$ then $\overline{\alpha} = p?a$ else $\overline{\alpha} = p!a$. This can be extended to sequences of actions.

Definition 22 *An IOLTS M is said*
• **deterministic** *if* $\forall \sigma \in (A^M)^*, |M \ after \ \sigma| \leq 1$ *where* $|X|$ *is the cardinality of the set* X*,*
• **controllable** *if in each state of M, either only one output is enabled or all inputs are enabled: for any sequence* $\sigma \in (A^M)^*$*,*
$In(M \ after \sigma) = A_I^M$ *or* $(In(M \ after \sigma) = \emptyset \wedge |Out(M \ after \sigma)| \leq 1)$*,*
• **input-complete** *if any input is possible after each trace:* $\forall \sigma \in (A^M)^*$, $In(M \ after \sigma) = A_I^M$.

As usual [10, 12], a **specification** of a system S will be modeled by an IOLTS $S = (Q^{\mathrm{S}}, P_I^{\mathrm{S}}, P_O^{\mathrm{S}}, A_I^{\mathrm{S}}, A_O^{\mathrm{S}}, T^{\mathrm{S}}, q_{\mathrm{init}}^{\mathrm{S}})$ and an **implementation** by a deterministic input-complete IOLTS $I = (Q^{\mathrm{I}}, P_I^{\mathrm{I}}, P_O^{\mathrm{I}}, A_I^{\mathrm{I}}, A_O^{\mathrm{I}}, T^{\mathrm{I}}, q_{\mathrm{init}}^{\mathrm{I}})$, with $P_I^{\mathrm{I}} = P_I^{\mathrm{S}}$, $P_O^{\mathrm{I}} = P_O^{\mathrm{S}}$, $A_I^{\mathrm{S}} \subseteq A_I^{\mathrm{I}}$, and $A_O^{\mathrm{S}} \subseteq A_O^{\mathrm{I}}$. A **test case** is a set of sequences of actions describing all the interactions occurring between an IUT and a tester which wants to verify that an IUT conforms with its specification. A test case is modeled by a deterministic IOLTS $T = (Q^{\mathrm{T}}, P_I^{\mathrm{T}}, P_O^{\mathrm{T}}, A_I^{\mathrm{T}}, A_O^{\mathrm{T}}, T^{\mathrm{T}}, q_{\mathrm{init}}^{\mathrm{T}})$ such that: $A_I^{\mathrm{T}} = A_O^{\mathrm{I}}$ (every possible output of the IUT must be considered as an input of the test case), $A_O^{\mathrm{T}} = A_I^{\mathrm{S}}$ (a test case should only send outputs that are waited by the specification), $\{pass, fail\} \in Q^{\mathrm{T}}$ with $enable(pass) = enable(fail) = \emptyset$, and the last transition leading to *fail* is an input. In general, it is assumed that a test case is controllable and input-complete.

Remark: In practice $A_I^{\mathrm{T}} = A_O^{\mathrm{I}}$ is unknown. Thus, only inputs not leading to *fail* can be denoted, the other inputs are implicitly leading to fail or are denoted by "? otherwise Fail", like in TTCN (see [9]-part 3). ⋄

2.2 Conformance

Formalizing conformance testing [4] necessitates to define a conformance relation relating models of IUTs to specifications. We will consider here a conformance relation which states that outputs produced by an IUT after a trace of the specification are foreseen by the specification [10, 12].

Definition 23 *(Conformance relation) Let S be the IOLTS describing the specification and I an (input-complete) IOLTS describing an implementation:* $I \; ioconf \; S \iff \forall \sigma \in Traces(S), Out(I \; after \; \sigma) \subseteq Out(S \; after \; \sigma)$.

Definition 24 *(Synchronous Testing) The synchronous application of a test case to an IUT is defined as a parallel composition* $\|$ *of the test case T and the IUT I:* $\forall \alpha \in A_I^T \cup A_O^T, \frac{T \xrightarrow{\alpha} T' \quad I \xrightarrow{\bar{\alpha}} I'}{T \| I \xrightarrow{\alpha} T' \| I'}$.

Definition 25 *(Test failure and unbiased test case)*
$T \; fails \; I =_{def} \exists \, I', \exists \sigma, T \| I \overset{\sigma}{\Rightarrow} fail \, \| \, I'$. *A test case T is unbiased with respect to S if and only if* $\forall I, T \; fails \; I \Rightarrow not(I \; ioconf \; S)$.

The definition of *ioconf* authorizes IUTs to diverge from the specification starting from unspecified inputs: the specification implicitly authorizes any behaviour in the IUT after an unspecified input. We may need to make these behaviours explicit by considering input-complete specification. Moreover for input complete specifications, *ioconf* has a very simple characterization as stated by the following proposition.

Proposition 21 *Let S and I be two input-complete IOLTS.*
We have $I \; ioconf \; S \iff Traces(I) \subseteq Traces(S)$

Proof: Suppose $I \; ioconf \; S$ and let $w \in Traces(I)$. Suppose $w \notin Traces(S)$, then w can be split in two sequences $w = w_1.w_2$ where w_1 is the maximal prefix of $w \in Traces(S)$. Let α be the first action in w_2. If α is an input, as S is input complete, $w_1.\alpha \in Traces(S)$. If α is an output, as $w_1 \in Traces(S)$ and $I \; ioconf \; S$, $\alpha \in Out(S \; after \; w_1)$, and $w_1.\alpha \in Traces(S)$ which contradicts the hypothesis. Thus, $Traces(I) \subseteq Traces(S)$. Thus in both cases w_1 is not maximal and proves that $w \in Traces(S)$.
The converse i.e. $Traces(I) \subseteq Traces(S) \Rightarrow I \; ioconf \; S$ is evident even for a non input-complete specification S. ∎

As we may have to deal with specifications which are not input complete, we may want to complete them in order to benefit from the preceding result. But this completion should not alterate the set of conformant IUTs. This is achieved by using the following completion similar to the notion of trap-state [13] and illustrated by figure 2.

Definition 26 *(Completion)*
Let S be an IOLTS. Comp(S) is defined as an IOLTS such that:
$Traces(Comp(S)) = Traces(S) \bigcup (\cup_{w \in Traces(S), a \in U(w)} w.?a.(\{?\} \times A_I \cup \{!\} \times A_O)^*)$
where $U(w) = A_I \setminus In(S \; after \; w)$ *is the set of unspecified inputs of the state of S reached after the trace w.*

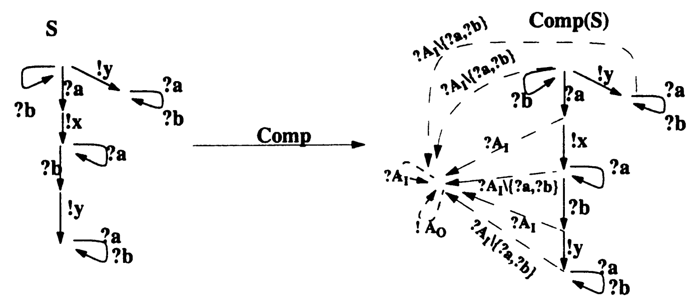

Figure 2 Example illustrating the completion (dotted lines represent transitions added by the transformation)

Proposition 22 *Let S be an IOLTS and $Comp(S)$ its input completion. For all input complete IOLTS I, I ioconf $S \Longleftrightarrow I$ ioconf $Comp(S)$.*

Proof: Suppose $\neg(I \textit{ ioconf } S)$. By definition of *ioconf*, this means $\exists\sigma \in Traces(S), \exists z \in Out(I \textit{ after } \sigma)$, and $z \notin Out(S \textit{ after } \sigma)$. By definition of *Comp*, we have $Traces(S) \subseteq Traces(Comp(S))$, so $\sigma \in Traces(Comp(S))$. But $\sigma.z \notin Traces(Comp(S))$ as $z \in A_O$ and the input completion only adds $a \in A_I \backslash In(S \textit{ after } \sigma)$. This implies $\neg(I \textit{ ioconf } Comp(S))$.
Suppose now $\neg(I \textit{ ioconf } Comp(S))$, this means $\exists\sigma \in Traces(Comp(S))$, $\exists z \in Out(I \textit{ after } \sigma)$, and $z \notin Out(Comp(S) \textit{ after } \sigma)$. If $\sigma \in Traces(S)$ then $z \notin Out(Comp(S) \textit{ after } \sigma)$ induces $z \notin Out(S \textit{ after } \sigma)$ as $Traces(S) \subseteq Traces(Comp(S))$. Otherwise $\sigma \in Traces(Comp(S)) \backslash Traces(S)$ is of the form $\sigma_1.a.\sigma_2$ with $\sigma_1 \in Traces(S)$ and $a \in A_I$. In this case, $\sigma_1.a.\sigma_2.z \in Traces(Comp(S))$, i.e. $z \in Out(Comp(S) \textit{ after } \sigma)$ which contradicts the hypothesis. ■

Remark: The validity of the proposition 22 heavily depends on the notion of completion used. For example completing with loops on unspecified inputs does not work.

There also exist different notions of input completion. For instance in SDL, input completion is implicit in the case of unspecified inputs. This means that in any control state, if an unspecified input occurs it is consumed. But as transitions are atomic, this happens only in any control (stable) states. Thus in terms of transition systems, only some states are input complete. The other states should be completed with *Comp* in order to apply 22.

Notice also that the notion of completeness of Mealy machines (see [8] for example) corresponds more to this implicit completeness of SDL than to the notion of input-completeness of definition 26. ◇

Propositions 21 and 22 lead to the following proposition 23 which gives a very simple characterization of *ioconf*.

Proposition 23 *I ioconf $S \Longleftrightarrow Traces(I) \subseteq Traces(Comp(S))$.*

3 REMOTE ASYNCHRONOUS TESTING

In practice, the testing activity is generally done through an environment intercalated between the tester and the IUT. As advocated in [4] the conformance of an IUT with respect to a specification in a context should be defined as the conformance of the implementation in its context with respect to the specification in a model of the context. For example, in the context of a remote testing architecture, the tester reaches the IUT through a network. In this case, the interaction between the tester and the IUT is asynchronous and PCOs can be seen as composed of two FIFO queues.

We first define the asynchronous transformation $\mathcal{A}$ on IOLTS which describes the impact of FIFO queues on observable behaviours. We use $\mathcal{A}$ to define the conformance relation in an asynchronous environment. We notice how this affects conformance: the set of conformant IUTs in synchronous or asynchronous architectures are not comparable in general. However, we show that for input-complete specifications, synchronous conformance implies asynchronous conformance.

3.1 Asynchronous testing

As already done in [14], we define the asynchronous transformation $\mathcal{A}$ as follows:

Definition 31 *Let* $M = (Q^M, P_I^M, P_O^M, A_I^M, A_O^M, T^M, q_{init}^M)$ *be an IOLTS.*
$\mathcal{A}(M) = (Q^{A(M)}, P_I^{A(M)}, P_O^{A(M)}, A_I^{A(M)}, A_O^{A(M)}, T^{A(M)}, q_{init}^{A(M)})$ *with:*
• $Q^{A(M)} = Q^M \times \prod_{p \in P_I^M} A_I^{M*} \times \prod_{p \in P_O^M} A_O^{M*}$ *and* $q_{init}^{A(M)} =< M, (\epsilon \cdots \epsilon), (\epsilon \cdots \epsilon) >$
• $P_I^{A(M)} = P_I^M$ *and* $P_O^{A(M)} = P_O^A$, $A_I^{A(M)} = A_I^M$ *and* $A_O^{A(M)} = A_O^M$,
• $T^{A(M)}$ *is described by the following operational rules defined for all* $q, q' \in Q^M, a \in A_I^M, b \in A_O^M, p_{Ik} \in P_I^M, p_{Ol} \in P_O^M$:

R1 (inputs of $\mathcal{A}(M)$ from Env)

$$< q, (p_{I1} \cdots p_{Ik} = w \cdots), (p_{O1} \cdots) > \overset{p_{Ik}?a}{\rightarrow}_{A(M)} < q, (p_{I1} \cdots p'_{Ik} = w.a \cdots), (p_{O1} \cdots) >$$

R2 (outputs of $\mathcal{A}(M)$ to Env)

$$< q, (p_{I1} \cdots), (p_{O1} \cdots p_{Ol} = b.w \cdots) > \overset{p_{Ol}!b}{\rightarrow}_{A(M)} < q, (p_{I1} \cdots), (p_{O1} \cdots p_{Ol} = w \cdots) >$$

R3 (internal actions)

$$\frac{q \overset{\tau}{\rightarrow}_M q'}{< q, (p_{I1} \cdots), (p_{O1} \cdots) > \overset{\tau}{\rightarrow}_{A(M)} < q', (p_{I1} \cdots), (p_{O1} \cdots) >}$$

R4(inputs of M from queues)

$$\frac{q \overset{p_{Ik}?a}{\rightarrow}_M q'}{< q, (p_{I1} \cdots p_{Ik} = a.w \cdots), (p_{O1} \cdots) > \overset{\tau}{\rightarrow}_{A(M)} < q', (p_{I1} \cdots p_{Ik} = w \cdots), (p_{O1} \cdots) >}$$

R5 (outputs of M to queues)

$$\frac{q \overset{p_{Ol}!b}{\rightarrow}_M q'}{< q, (p_{I1} \cdots), (p_{O1} \cdots p_{Ol} = w \cdots) > \overset{\tau}{\rightarrow}_{A(M)} < q', (p_{I1} \cdots), (p_{O1} \cdots p_{Ol} = w.b \cdots) >}$$

Remark: $\forall$ *IOLTS* $M, Traces(M) \subseteq Traces(\mathcal{A}(M))$. ◇

For the asynchronous context the conformance relation now becomes:

Definition 32 *(Conformance in an asynchronous environment) Let I and S be two IOLTS with I input-complete. I* $ioconf_{\mathcal{A}}$ $S =_{def} \mathcal{A}(I)$ *ioconf* $\mathcal{A}(S)$.

The notion of test failure and unbias in an asynchronous environment are then straightforward.

3.2 Problems in asynchronous testing

Let us consider the specification S described on the left part of figure 3. For sake of clarity, we will suppose that all the indicated observable actions occur on the same PCO. Notice also that S is not input-complete. The right

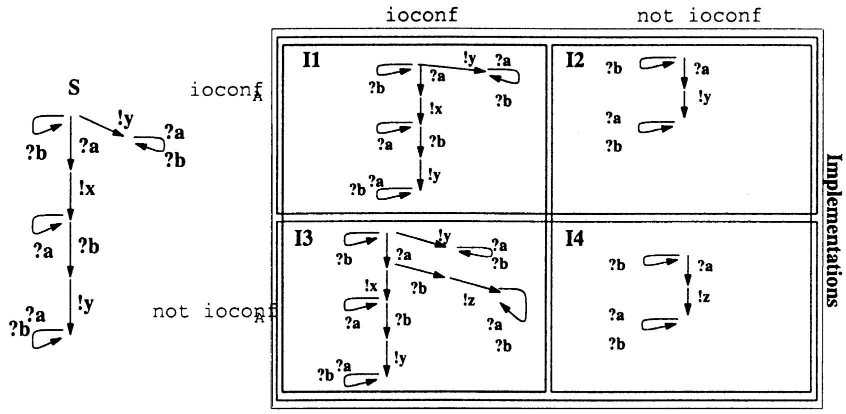

Figure 3 Difference between synchronous and asynchronous testing

part of figure 3 contains different implementations which show that testing synchronously or asynchronously is not comparable. More precisely, they show two main problems when testing in an asynchronous environment.

Permissiveness: The IUT I_2 shows that asynchronous testing is more permissive than synchronous testing: $\neg(I_2$ *ioconf* $S)$ but $(I_2$ $ioconf_{\mathcal{A}}$ $S)$. We have $\neg(I_2$ *ioconf* $S)$ because $Out(I_2$ *after* $?a) = \{y\} \not\subseteq Out(S$ *after* $?a) = \{x\}$. In an asynchronous environment, outputs can be delayed. Thus the trace $!y.?a$ of S can be observed as $?a.!y$ in $\mathcal{A}(S)$. Thus, we have $(I_2$ $ioconf_{\mathcal{A}}$ $S)$ as $Out(\mathcal{A}(I_2)$ *after* $?a) = \{y\} \subseteq Out(\mathcal{A}(S)$ *after* $?a) = \{x, y\}$.

Non preservation of conformance: This problem is brought to light by the IUT I_3: I_3 *ioconf* S but $\neg(I_3$ $ioconf_{\mathcal{A}}$ $S))$. In fact, I_3 *ioconf* S even though $?a.?b.!z$ is not a trace of S. This is because *ioconf* authorizes divergence from the specification starting from an unspecified input. The sequence

$?a.?b$ is in $Traces(\mathcal{A}(S))$, and we have $Out(\mathcal{A}(S)$ *after* $?a.?b) = \{x, y\}$ but $Out(\mathcal{A}(I_3)$ *after* $?a.?b) = \{x, y, z\}$. Thus $\neg(I_3$ *ioconf*$_A$ $S)$.

Notice that these problems are more complex in the context of several PCOs as inputs and outputs orders are not preserved. Similar remarks concerning the non preservation of conformance in asynchronous environment have been done in [14]. But these were made regarding the synchronous conformance relation *conf* (which does not distinguish inputs and outputs) and an asynchronous conformance relation similar to *ioconf*$_A$.

The problem of permissiveness is inherent to the transformation by a context. But the non preservation of conformance is due to the fact that S is not input-complete. Lemma 31 states that this can be avoided input-complete specifications and **monotonic** transformations by contexts. A transformation $\mathcal{T}$ on IOLTS is **monotonic** if it preserves trace inclusion:
$Traces(M) \subseteq Traces(M') \Rightarrow Traces(\mathcal{T}(M)) \subseteq Traces(\mathcal{T}(M'))$.

Lemma 31 *Let S and I be input complete IOLTS. If $\mathcal{T}$ is a monotonic transformation, I ioconf $S \Rightarrow \mathcal{T}(I)$ ioconf $\mathcal{T}(S)$.*

Proof: Suppose I *ioconf* S and S and I are input complete. By proposition 21 we have $Traces(I) \subseteq Traces(S)$. By monotonicity of $\mathcal{T}$, this implies $Traces(\mathcal{T}(I)) \subseteq Traces(\mathcal{T}(S))$ which implies $\mathcal{T}(I)$ *ioconf* $\mathcal{T}(S)$. ■

The asynchronous transformation $\mathcal{A}$ is monotonic because it is applied on traces independently of others. Application of lemma 31 gives the corollary:

Corollary 31 *For input complete IOLTS S and I, I ioconf $S \Rightarrow I$ ioconf$_A$ S.*

As a consequence, the only difference between *ioconf* and *ioconf*$_A$ is permissiveness. We show in the next section that this can be avoided if the order of occurrence of events on the IUT is captured by an appropriate stamp mechanism.

4 STAMPED ASYNCHRONOUS TESTING

The idea is to instrument the IUT so that each output from the IUT to the tester (via the environment) can bring to the tester an additional information on the real order in which the IUT has produced the events. Linking time stamp techniques used in observing distributed systems to the problem of generating tests has been already advocated in different contexts like in [6, 7].

This instrumentation can be defined by the synchronous parallel composition of the IUT with a stamp process ST (as illustrated in figure 4). The role of the stamp process ST is to code the history of occurrence of events on the IUT and to transmit it to the environment by piggybacking each output.

We will consider the particular case of one PCO in which history is coded by an integer counter.

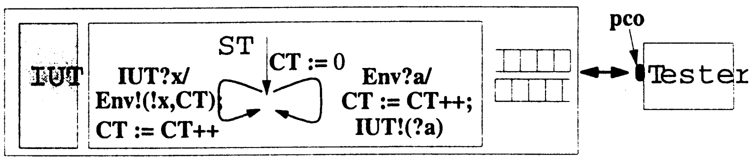

Figure 4 Implementation of the counting mechanism *st*

This stamping process ST implements a function *st* defined on traces as follows.

Definition 41 $st : (A_I \cup A_O)^* \to (A_I \cup (A_O \times \mathbb{N}))^*$, $\forall \sigma \in (A_I \cup A_O)^*, a \in A_I$, *and* $z \in A_O$: $st(\epsilon) = \epsilon$, $st(\sigma.!z) = st(\sigma).(!z, length(\sigma))$, *and* $st(\sigma.?a) = st(\sigma).?a$.

By extension, one can regard the transformation *st* as being carried out on the IOLTS by an on-the-fly traversal of the graph. Let us consider the example of an interaction between an IUT and its environment illustrated on figure 5. From the point of view of events produced on the IUT, one has: $st(?x.!s.?y.?t.?u.!z) = ?x.(!s,1).?y.?t.?u.(!z,5)$.

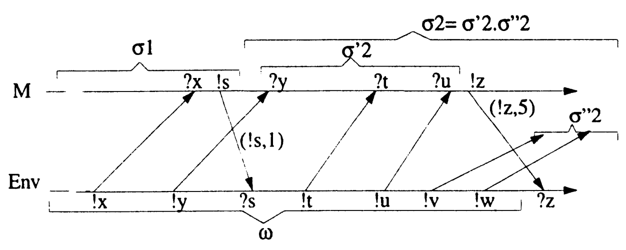

Figure 5 Illustration of the computation of st and $\overline{st}$

This counting information is intended to be decoded by the tester as described in section 5. We thus define an opposite transformation $\overline{st}$ (illustrated in figure 5), which orders events of a system M in the environment. From a sequence of events ω in the environment *Env*, $\overline{st}$ computes two sequences of events $\overline{st}(\omega)[1]$ and $\overline{st}(\omega)[2]$ using the counting information associated to outputs of the system M:

- $\overline{st}(\omega)[1]$ is the sequence of events of the system M which precede the last output z of M received by the environment and including this output.
- $\overline{st}(\omega)[2]$ is the sequence of inputs of M (corresponding to outputs of *Env*) which were not received before the sending by M of z. As we assume a FIFO channel between *Env* and M, this sequence will be received in this order but M may have some intercalated outputs of M.

Notice that $\overline{st}$ can be computed on-the-fly by *Env*. When *Env* makes an output, the corresponding input is added to the tail of $\overline{st}(\omega)[2]$. When *Env* makes an input z with the counting information, $\overline{st}(\omega)[1]$ and $\overline{st}(\omega)[2]$ are updated accordingly.

Definition 42 *$\overline{st} : (A_I \cup (A_O \times \mathbb{N}))^* \rightarrow (A_I \cup A_O)^* \times A_I^*$ is defined recursively as: $\overline{st}(\epsilon) = (\epsilon, \epsilon)$, and $\forall \omega \in (A_I \cup (A_O \times \mathbb{N}))^*$, if $\overline{st}(\omega) = (\overline{st}(\omega)[1], \overline{st}(\omega)[2])$ then $\overline{st}(\omega.?a) = (\overline{st}(\omega)[1], \overline{st}(\omega)[2].?a)$ and $\overline{st}(\omega.(!z, i)) = (\overline{st}(\omega)[1].\sigma_2'.!z, \sigma_2'')$ where σ_2' and σ_2'' are sequences such that $\overline{st}(\omega)[2] = \sigma_2'.\sigma_2''$ with $length(\sigma_2') = i - length(\overline{st}(\omega)[1])$.*

For example, let us consider the sequence of events occurring on the tester in figure 5 (denoted by their corresponding names on the IUT). Then we have $\overline{st}(?x.?y.(!s,1).?t.?u.?v.?w) = (?x.!s,\ ?y.?t.?u.?v.?w)$, and $\overline{st}(?x.?y.(!s,1).?t.?u.?v.?w.(!z,5)) = (?x.!s.?y.?t.?u.!z,\ ?v.?w)$. In this last case, this means that upon reception of z, the tester knows that $?x.!s.?y.?t.?u.!z$ has occurred in this order on the IUT, but it does not know yet what will be the order in the future including the receptions of v and w. The counting information gives the index for inserting the output in the sequence.

The following proposition states that the composition of st with the asynchronous transformation $st \circ \mathcal{A}$ is invertible: for a trace σ, the application of $\overline{st}$ on any trace of $st \circ \mathcal{A}(\sigma) = \mathcal{A}(st(\sigma))$ can reconstruct σ.

Proposition 41 *$\forall M \in IOLTS, \forall \sigma \in Traces(M), \forall \omega \in Traces(\mathcal{A}(st(\sigma)))$, we have: $\overline{st}(\omega) = (\sigma_1, \sigma_2)$ with $\sigma = \sigma_1.\sigma_2$.*

Proof: *st* associates with each output the index of this output in the sequence. This information can be used by $\overline{st}$ to recover the order since the transformation $\mathcal{A}$ (in the case of only one FIFO queue in each direction) preserves the relative order of inputs and the relative order of outputs. Notice that outputs do not need to be FIFO ordered as they are numbered by *st*. ∎

We can now prove the main theorem which says that for input complete IOLTS communicating asynchronously with their environment using one FIFO in each direction, remote asynchronous testing using stamps has the same testing power as local synchronous testing.

Theorem 41 *Let S and I be two input complete IOLTS communicating asynchronously with their environment using one FIFO in each direction.
We have: $I\ ioconf\ S \iff st(I)\ ioconf_A\ st(S)$*

Proof: We first prove that $I\, ioconf S \Rightarrow \mathcal{A}(st(I))\, ioconf\, \mathcal{A}(st(S))$. Since $\mathcal{A}$ and *st* are monotonic (*st* is defined on traces), $st \circ \mathcal{A}$ is also monotonic and lemma 31 applies.

Let us prove the converse: if $\neg(I\ ioconf S)$ then $\neg(\mathcal{A}(st(I))\ ioconf\, \mathcal{A}(st(S)))$. By definition of *ioconf*, $\neg(I\ ioconf S) \Rightarrow \exists \sigma \in Traces(S), \exists z \in Out(I after \sigma)$ such that $z \notin Out(S after \sigma)$. Let $\omega = st(\sigma)$. For any $IOLTS\ M, \sigma \in Traces(M)$ implies $st(\sigma) \in Traces(st(M))$ and $Traces(M) \subseteq Traces(\mathcal{A}(M))$. Thus, we

have: $\omega \in Traces(\mathcal{A}(st(S)))$ and $\omega.(z, length(\sigma)) \in Traces(\mathcal{A}(st(I)))$. We want to prove that $\omega.(z, length(\sigma)) \notin Traces(\mathcal{A}(st(S)))$.
Since $z \notin Out(S after \sigma)$, we must show that the output $(z, length(\sigma))$ after ω cannot be created by the $st \circ \mathcal{A}$ transformation. First, note that the traces of $\mathcal{A}(\sigma)$ are produced from σ by the following semi-commutation $!x.?y \to ?y.!x$ which can only delay the outputs with respect to inputs. Let us suppose that z belongs to a trace σ' of S. We show that σ' is identical to σ. $\forall \sigma' \in Traces(S), \exists \omega \in \mathcal{A}(st(\sigma'))$ and $\omega = st(\sigma) \Rightarrow \sigma = \sigma'$. This is based on the properties of $\mathcal{A}$: a) $length(\sigma) = length(\sigma')$, b) σ and σ' are on the same alphabet, c) the outputs are numbered by the same stamps, d) the inputs are not changed by the transformation.
Thus, $\exists \omega \in Traces(\mathcal{A}(st(S))), \omega.(z, length(\sigma)) \notin Traces(\mathcal{A}(st(S)))$ which implies $\exists \omega \in Traces(\mathcal{A}(st(S))), (z, length(\sigma)) \in Out(\mathcal{A}(st(I)\ after\ \omega)$, and $(z, length(\sigma)) \notin Out(\mathcal{A}(st(S)\ after\ \omega)$. ■

Stamped asynchronous tests generation Theorem 41 induces a method for the generation of test cases for asynchronous testing with the power of synchronous testing.

Proposition 22 says that for any specification S its input completion $Comp(S)$ has the same set of conformant implementations. Theorem 41 says that if we assume a communication with the environment with one FIFO in each direction, an IUT I is conformant to $Comp(S)$ if and only if $\mathcal{A}(st(I))$ is conformant with $\mathcal{A}(st(Comp(S)))$. Now, there exists test generation algorithms implemented in tools [3, 15]. Applied to a specification S, they produce unbiased test cases with respect to S and *ioconf*. These methods are (theoretically) exhaustive: assuming bounded fairness of implementations, all non conformant implementations may be rejected by a test case. Using these algorithms on $\mathcal{A}(st(Comp(S)))$ will thus produce a test suite which is unbiased and exhaustive with respect to $\mathcal{A}(st(Comp(S)))$ and *ioconf*. Moreover, by theorem 41 this test suite has exactly the same testing power as the synchronous test suite.

But, generating tests from $\mathcal{A}(st(Comp(S)))$ has some drawbacks. The first one is the state space explosion due to the asynchronous transformation and the unfolding caused by st. A second one is the relevance of test cases. In the example of figure 3, $?a.?b.(!x, 1)$ and $?a.(!x, 1).?b$ are both sequences of $\mathcal{A}(st(Comp(S)))$. So one could generate a test case with a sequence $!a.!b.(?x, 1)$. But this would be artificial because as S sends x after the input a, it is preferable to wait for x before sending b.

So, when testing the conformance of I with respect to S in a remote testing architecture, we are mainly interested in traces of S but not in all traces of $\mathcal{A}(st(Comp(S)))$, even if we have to consider them as possible traces. One way to limit these drawbacks is thus to use test purposes which accept traces in $Traces(S)$ in order to select test cases from $\mathcal{A}(st(Comp(S)))$ and to generate test cases on-the-fly as in [3]. Another completely different and more efficient solution is investigated in the following section.

5 REMOTE ASYNCHRONOUS TESTING WITH SYNCHRONOUS TEST CASES

The idea is that from a sequence of events of $\mathcal{A}(st(I))$, $\overline{st}$ can reconstruct, with a certain latency due to the asynchronous communication, the sequence of events which occurred on I. Thus this sequence can be checked on test cases produced from S to check the conformance of I with respect to S and *ioconf* as in a local synchronous testing architecture. Thus local synchronous test cases are generated from $Comp(S)$. Then, test cases are decorated with counters using the transformation st. They can then be played in a remote asynchronous testing architecture with a kind of test driver implementing $\overline{st}$. This is illustrated in figure 6.

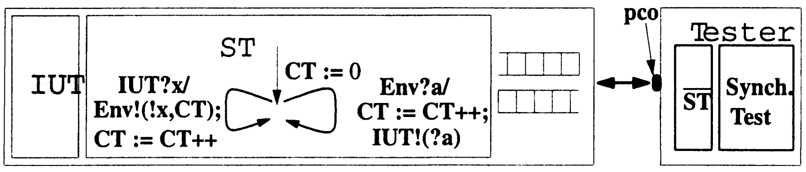

Figure 6 Architecture for "remote local" testing

The mechanism necessitates some attention. In fact the difference between synchronous local testing and asynchronous testing is that the control of the tester on the IUT is weakened. Inputs of the IUT cannot be completely controlled by the tester. This can be illustrated on the example of figure 3. According to the specification, the tester may choose an output (say a) and computes $\overline{st}(?a) = (\epsilon, ?a)$. But between the output $!a$ by the tester and its corresponding input $?a$ by the IUT, the IUT may decide to perform the output $!y$ associated with the counter 0. When receiving $(y, 0)$, $\overline{st}(?a.(!y, 0)) = (!y, ?a)$. Thus the tester knows that the IUT has performed $!y$ first and will later receive a. The IUT has thus chosen a different behaviour from the one chosen by the tester. Nevertheless, the tester must evolve according to the behavior of the IUT but also to its own past behavior. In particular, a has been sent and this cannot be cancelled. Thus the tester will have to wait for a new input from the IUT or choose a new output according to the sequence $!y.?a$. All this information is contained in $\overline{st}(?a.(!y, 0))$.

The tester thus computes $\overline{st}(\omega)$ on-the-fly. Only $\overline{st}(\omega)[1]$ should be used for verdicts because it is a sequence of I. The information on the sequence of inputs of the IUT of $\overline{st}(\omega)[2]$ is not complete as outputs can be intercalated in the sequence. The tester may sometimes choose to wait for an output of the IUT to complete its information. But this is not always possible because the IUT may also wait for an input. So outputs of the tester must be chosen and this is done according to $\overline{st}(\omega)[1].\overline{st}(\omega)[2]$.

A fundamental difference with synchronous local testing is that test cases are controllable in synchronous local testing: a test case never has the choice between two outputs or between an input and an output. This comes from the fact that the tester controls the inputs of the IUT. This is not anymore the case when synchronous test cases are played in an asynchronous environment. In this case, in any state all possible outputs of the IUT must be taken into account. The controllability property thus has to be relaxed into **semi-controllability**. An IOLTS is said **semi-controllable** if in any state at most one output is possible and all possible inputs are considered:
$\forall \sigma \in (A^{\text{M}})^*$, $In(M\ after\,\sigma) = A_I^{\text{M}} \wedge |Out(M\ after\,\sigma)| \leq 1$.

Also specifications have to be input complete or completed by *Comp*. In fact if $\overline{st}(\omega)[1] = \sigma.!z \in Traces(I)$, either $z \notin Traces(S\ after\ \sigma)$ and in this case $!z$ should produce a fail verdict, or $z \in Traces(S\ after\ \sigma)$ and $\overline{st}(\omega)[1].\overline{st}(\omega)[2]$ must be in $Traces(S)$. As $\overline{st}(\omega)[2]$ is composed of inputs, input completeness always ensures this.

Notice that the first component of $\overline{st}$ is monotonic for the prefix ordering i.e. $\omega \leq \omega'$ for the prefix ordering implies $\overline{st}(\omega)[1] \leq \overline{st}(\omega')[1]$.

The algorithm which has to be performed on the test driver is described below. It uses two sequences σ_1 and σ_2 which respectively contain the two components of the sequence of events ω performed by the tester: $\sigma_1 = \overline{st}(\omega)[1]$ and $\sigma_2 = \overline{st}(\omega)[2]$. The variable ω is only used to describe the invariant.

```
■ Input: TC: test case. Output: verdict
  Invariant $\sigma_1 = \overline{st}(\omega)[1]$ and $\sigma_2 = \overline{st}(\omega)[2]$
  Initialization: $\omega := \epsilon$; $\sigma_1 := \epsilon$; $\sigma_2 := \epsilon$; verdict:= nil
  (* $\overline{st}(\epsilon) = (\epsilon, \epsilon)$ *)
  while verdict = nil do
      non-deterministic choice
         if $Out(TC\ after\,\sigma_1.\sigma_2) \neq \emptyset$
             send(a) where $\{a\} = Out(TC\ after\,\sigma_1.\sigma_2)$
           (* a is unique as test cases are semi-controllable *)
             $\omega := \omega.?a$; $\sigma_2 := \sigma_2.?a$; (* i.e. $\overline{st}(\omega.?a) = (\overline{st}(\omega)[1], \overline{st}(\omega)[2].?a$*)
         if input queue is not empty
             receive($(z,i)$); $\omega = \omega.(!z,i)$
             let $\sigma_2', \sigma_2''$ s.t. $length(\sigma_1.\sigma_2') = i$ and $\sigma_2 = \sigma_2'.\sigma_2''$
             $\sigma_1 := \sigma_1.\sigma_2'.!z$; $\sigma_2 := \sigma_2''$; (* $\overline{st}(\omega.(!z,i)) = (\overline{st}(\omega)[1].\sigma_2'.!z, \sigma_2'')$ *)
            verdict := verdict assigned to the state $(TC\ after\,\sigma_1)$
  end
```

6 CONCLUSION AND PERSPECTIVES

We have described how remote asynchronous testing with counters could have the same testing power as local synchronous testing in the case where the communication between the IUT and the environment is done through one PCO. This can be generalized to multiple PCOs but needs more sophisticated mechanisms. The general idea is that the tester needs to reorder events of the IUT in order to reach the same testing power as in the synchronous case. It is natural to suppose that the IUT can only transmit information to the tester by piggybacking its outputs. In order to emit a correct verdict on an output of the IUT, the tester needs to know the whole sequence of events of the IUT which precedes this output (see figure 7). Thus a theoretical instrumentation consists in associating to each output of the IUT the sequence of its predecessors in the IUT.

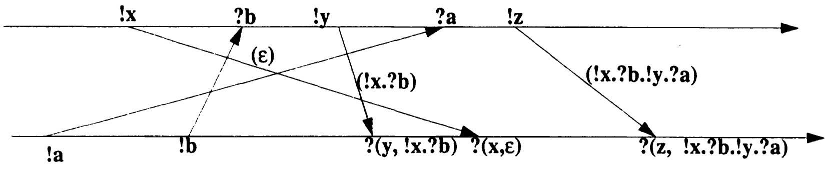

Figure 7 Theoretical instrumentation in the general case

This instrumentation is not realistic because this induces redundant information and the piggybacked sequences continuously grow. A non redundant instrumentation consists in segmenting the information into sub-sequences between two consecutive outputs. But as an output can be overtaken by a following one, the order of this output in the sequence should be piggybacked as in the case of one PCO. The functions *st* and $\overline{st}$ must be extended in the case of multiple PCOs.

Another prospect is timers management. Deadlocks and output quiescence are considered as outputs in the testing theory [12]. In our framework of asynchronous instrumented testing, inputs of test cases (outputs of the IUT) carry counters. This must be generalized to timeouts but suggests that timers should be managed by the instrumentation of the IUT. Moreover starting and cancelling a timer must be done by the instrumentation but initiated by the tester with messages.

REFERENCES

[1] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification*, North Holland, 1988. IFIP Transactions.

[2] L. Doldi, V. Encontre, J.-C. Fernandez, T. Jéron, S. Le Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In B. Baumgarten, H.-J. Burkhardt,

and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems.* Chapman & Hall, p. 346–361, September 1996.

[3] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In A. Alur and T. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, LNCS 1102. Springer, July 1996.

[4] FMCT ISO/IEC JTC1/SC21 WG7, ITU-T. SG 10/Q.8. Information Retrieval, Transfer ad Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500. ISO - ITU-T, Geneve, 1996.

[5] ITU-T Z.100. Specification and Description Language (SDL). Recommendation ITU Z.100, 1996.

[6] C. Jard. How to observe interoperability at the service layer of protocols. In T. Mizuno, T Higashino, and N. Shiratori, editors, *IFIP TC6 7th International Workshop on Protocol Test Systems.* Chapman & Hall, p. 259–270, November 1994.

[7] M. Kim, S.T. Chanson, S. Kang, and J. Shin. An approach for testing asynchronous communicating systems. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems.* Chapman & Hall, p. 141–155, September 1996.

[8] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A survey. In Proceedings of the IEEE, vol. 84(8), p. 1090–1123, August 1996.

[9] OSI Open Systems Interconnection. Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1996.

[10] M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties.* PhD thesis, Université de Bordeaux, 1994.

[11] J. Tretmans. *A formal approach to conformance testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[12] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools.* Springer-Verlag, vol. 17, No. 3, p. 103–120, 1996.

[13] S. H. Unger. *Asynchronous sequential switching circuits.* John Wiley and Sons, 1969.

[14] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On Asynchronous Testing. In G. Von Bochman, R. Dssouli, and A. Das, editors, *Fifth International Workshop on Protocol Test Systems*, North Holland, 1993. IFIP Transactions.

[15] R.G. de Vries, and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In Holzmann, G. and Najm, E. and Serhrouchni, A., editors, *Fourth Workshop on Automata Theoretic Verification with the SPIN Model Checker.* ENST 98 S 002, p. 115-128, November 1998, Paris, France.