Automatic Interoperability Test Case Generation based on Formal Definitions

Alexandra Desmoulin and César Viho

IRISA/Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France
{alexandra.desmoulin,viho}@irisa.fr

Abstract. The objective of this study is to provide methods for deriving automatically interoperability tests based on formal definitions. First, we give interoperability formal definitions taking into account both objectives of interoperability: the implementations must interact correctly and the expected service must be provided. Based on these definitions, a method for generating interoperability test cases is described. This method is equivalent to classical methods in terms of non-interoperability detection and avoids state-space explosion problem. Classical and proposed methods were implemented using the CADP Toolbox and applied on a connection protocol to illustrate this contribution.

1 Introduction

In the domain of network protocols, implementations are tested to ensure that they will correctly work in an operational environment. These implementations are developed based on specifications, generally standards. Different kinds of tests exist. Among these tests, conformance and interoperability testing considers the behaviour of these implementations at their interfaces to verify that they will work correctly in a real network. The aim of conformance tests [1] is to verify that a single implementation behaves as described in its specification. Interoperability testing objectives are to check both if different implementations communicate (or interact) correctly and if they provide the services described in their respective specification during this interaction.

Conformance testing has been formalized [1, 2]. A formal framework exist together with testing architectures, formal definitions and methods for writting conformance tests. Moreover, this formalization leads to automatic test generation methods and tools like TGV [3] or TorX [4].

In the interoperability testing situation, there is no precise characterization of interoperability for the moment. However, some attemps to give formal definitions of interoperability exists [5, 6]. Some studies also give methods for automatically deriving interoperability tests [7, 8]. These methods are generally based on fault models or on the search of some particular kinds of error. But there is still no method for the moment describing how to generate interoperability test cases based on formal definitions.

In a previous study [5], we give some formal definitions of the notion of interoperability together with some clues for test generation. In this study, we

complete these formal definitions to take into account all the objectives of interoperability testing. Thus, the so-called "interoperability criteria" describe the conditions that two implementations must satisfy to be considered interoperable and follow the main objectives of interoperability: providing the expected service while interacting correctly. Based on these definitions and using the clues given in [5], we develop a complete interoperability test derivation method. Moreover, we implement this method using the CADP Toolbox. Its application on a connection protocol allows us to verify both that it can generate interoperability test case equivalent to test cases that would have been obtained manually or with classical methods, and that it avoids state-space explosion problem that generally occurs with classical methods [6].

This paper is decomposed as follows. First, Section 2 describes notions used in the paper including interoperability testing architectures and formal models. Then, Section 3 is aimed at providing formal definitions of interoperability. Section 4 describes a method for deriving automatically interoperability tests based on these definitions. Section 5 describes the results of the application of the proposed interoperability test generation method on a connection protocol. Finally, conclusion and future work are in Section 6.

2 Preliminaries

In this Section, we present the different notions that are used in this study. First, we define interoperability testing and interoperability testing architecture in Section 2.1. Then, we describe the model of IOLTS used for interoperability formal definitions in Section 2.2. The proposed method for interoperability test generation reuses some aspects of conformance testing. Few words are said in Section 2.3 on the state of the art in automatic test generation.

2.1 Interoperability testing

We consider here the context of protocol testing. Protocol implementations are developped based on specifications, generally protocol standards. They must be tested to ensure that thay will work correctly in an operational environment. We consider here the context of black-box testing: the implementations are known by the events executed on their interfaces, generally sending and receiving messages. Among the different kinds of protocol testing contexts, we consider here interoperability testing that puts in relation different implementations (generally from different manufacturers) to verify that they are able to work together.
Interoperability testing has two goals. It verifies that different protocol implementations can communicate correctly, *and* that they provide the services described in their respective specification while communicating. In this study, we consider a context with two implementations under test (IUT for short): this is the one-to-one context (see Figure 1). In an interoperability testing architecture [9, 10], we can differentiate two kinds of interfaces: Lower Interfaces (used

for the interaction) and Upper Interfaces (used for the communication with upper layer). Testers are connected to these interfaces but they can *control* (send message) only the upper interfaces. The lower interfaces are only *observable*.

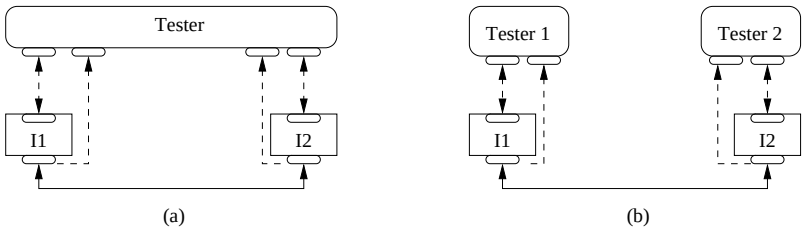

Fig. 1. Interoperability testing architectures

Depending on the access to the interfaces, different architectures can be distinguished. For example, the interoperability testing architecture is called *unilateral* if only the interfaces of one IUT are accessible during the interaction, *bilateral* if the interfaces of both IUTs are accessible but separately (Figure 1(b)), or *global* if the interfaces of both IUTs are accessible with a global view (Figure 1(a)).

2.2 IOLTS model

We use IOLTS (Input-Output Labeled Transition System) to model specifications. As usual in the black-box testing context, we also need to model IUTs, even though their behaviors are unknown. They are also modeled by an IOLTS.

Definition 1. *An IOLTS is a tuple* M=(Q^M,Σ^M,Δ^M, q_0^M). Q^M *is the set of states and* $q_0^M \in Q^M$ *the initial state.* Σ^M *denotes the set of observable events on the interfaces:* $p?m \in \Sigma^M$ *(resp.* $p!m \in \Sigma^M$*) stands for an input (resp. output) where* p *is the interface and* m *the message.* Δ^M *is the transition relation.*

Other Notations. Σ^{M} can be decomposed: $\Sigma^{\mathrm{M}} = \Sigma_U^{\mathrm{M}} \cup \Sigma_L^{\mathrm{M}}$, where Σ_U^{M} (resp. Σ_L^{M}) is the set of messages exchanged on the upper (resp. lower) interfaces. Σ^{M} can also be decomposed to distinguish input (Σ_I^{M}) and output messages (Σ_O^{M}). Based on this model, $Traces(q)$ is the set of executable traces (successions of events) from the state q. $\Gamma(q)$ is the set of executable events (on the interfaces of M) from the state q and $\Gamma(M,\sigma)$ the set of executable events for the system M after the trace σ. In the same way, $Out(M,\sigma)$ (resp. $In(M,\sigma)$) is the set of possible outputs (resp. inputs) for M after the trace σ. Considering a link between lower interfaces l_i of M_i and l_j of M_j, we also define $\bar{\mu}$ as $\bar{\mu}$=$l_i!a$ if $\mu = l_j?a$ and $\bar{\mu} = l_i?a$ if $\mu = l_j!a$.

Quiescence. An implementation can be *quiescent* in three different situations: either the IUT can be waiting on an input, either it can be executing a loop of

internal (non-observable) events, or it can be in a state where no event is executable. For an IOLTS M_i, a quiescent state q is modeled by $(q, \delta(i), q)$ where $\delta(i)$ is treated as an observable output event (practically with timers). The IOLTS M with quiescence modeled is called suspensive IOLTS and is noted $\Delta(M)$.

Interaction and projection. To give a formal definition of interoperability, two operations need to be modeled: asynchronous interaction and projection.

The *asynchronous interaction* is used to calculate the behavior - modeled by an IOLTS - of a system composed by two communicating entities. For two IOLTS M_1 and M_2, this interaction is noted $M_1 \|_{\mathcal{A}} M_2$. The way to obtain $M_1 \|_{\mathcal{A}} M_2$ is described in [5]. First, M_1 and M_2 are transformed into IOLTS representing their behavior in an asynchronous environment (as in [11, 12]). Then, these two IOLTS are composed to obtain $M_1 \|_{\mathcal{A}} M_2$ via the rules usually used for synchronous interaction. These rules (see for example [6, 13]) are "mapping" events on lower interfaces and propagating quiescence and events on upper interfaces.

The *projection* of an IOLTS on a set of events is used to represent the behavior of the system reduced to some events (such as events observable on specific interfaces). For example, the projection of M on the set of events executable on its lower interfaces Σ_L^M is noted M/Σ_L^M. It is obtained by hiding events (replacing by τ-transitions) that do not belong to Σ_L^M, followed by determinization. In the same way, $Out_X(M, \sigma)$ corresponds to a projection of the set of outputs $Out(M, \sigma)$ on the set of events X.

2.3 State of the art in automatic test generation

Some methods for generating automatically interoperability tests exists in [7, 14, 15, 16, 17]. However, these methods are not based on formal definitions. On the contrary, conformance testing is a kind of test for which a formal framework was developed. It determines to what extent a single implementation of a standard conforms to its requirements. Conformance testing architectures and formal definitions [1, 2] were described. Among these formal definitions, the **ioco** conformance relation [2] says that an implementation I is **ioco**-conformant to a specification S if I can never produce an output which could not be produced by its specification S after the same trace. Moreover, I may be quiescent only if S can do so. Formally : I **ioco** $S = \forall \sigma \in Traces(\Delta(S)), Out(\Delta(I), \sigma) \subseteq Out(\Delta(S), \sigma)$. This relation is the most used in practice for conformance tests. Defining formally conformance also allows automatic conformance test generation: conformance test generation tools like TGV [3] or TorX [4] are based on ioco-theory. Even though conformance and interoperability are two different kinds of test, they have in common to be based on traces of the specifications. Thus, part of the existing concepts of conformance testing can be reused for interoperability testing. However, the ioco-theory does not fit all objectives of interoperability testing (verification that the implementations communicate correctly and that they provide the expected services while interacting: see Section 3.2).

3 Formalizing interoperability

3.1 Specification model

As we are concerned with interoperability testing, the considered specifications must allow interaction. We call this property the interoperability specification compatibility property (iop-compatibility for short). Two specifications are iop-compatible iff, for each possible output on the interfaces used for the interaction after any trace of the interaction, the corresponding input is foreseen in the other specification. Formally, $\forall \sigma \in Traces(S_1 \| _{\mathcal{A}} S_2)$, $\forall \sigma.a.\sigma' \in Traces(S_1 \| _{\mathcal{A}} S_2)$, $a \in Out_{\Sigma_L}(S_1 \| _{\mathcal{A}} S_2, \sigma)$, $\sigma' = \beta_1 ... \beta_l$, $\Rightarrow \exists \beta_i$ such that $\beta_i = \bar{a}$. Practically, this property can be verified by a parallel search of both specifications -without constructing the specification interaction. This means that the traces of one specification must be compatible with possible execution of the other specification. Notice that this notion of iop-compatibility is different from the one described in [18] where authors consider that "two components are compatible if there is *some* environment that make them work together, simultaneously satisfying both of their environment assumption".
In some situations (underspecification of input actions particularly), the two specifications need to be completed to verify this property. It is done by adding transitions leading to an error trap state and labeled with the inputs corresponding to messages that can be sent by the interacting entity (input m added in $In(S_j, \sigma/\Sigma_j)$ if $\bar{m} \in Out_{\Sigma_L}(S_i, \sigma/\Sigma_i)$). Indeed, this method considers the reception of an unspecified input as an error. This is the most common definition of unspecified inputs in network protocols. In the following, we will consider that specifications are iop-compatible.

3.2 Formalization of interoperability principles

The purpose of interoperability testing is to verify that the two interconnected implementations communicate successfully **and** that they provide the expected services during their interaction. Interaction verification corresponds to the verification that outputs sent by an IUT on its lower interfaces are foreseen in the specification and that the interacting IUT is able to receive these messages. Service verification corresponds to the verification that outputs (and quiescence) observed on the upper interfaces of the IUTs are described in their respective specification. Thus, outputs must be verified on both upper and lower interfaces, while inputs are managed on lower interfaces.

Output verification is done by comparing output observed, after a particular trace, on the interfaces of the IUTs with the outputs foreseen in the specifications after the same trace. This part of interoperability testing can reuse ioco-theory. However, during test execution, there is an important difference between interoperability and conformance testing context also for output verification. Indeed, the lower interfaces are controllable in conformance context but during interoperability tests, these interfaces are only observable.

One of interoperability testing purposes is to verify that the two implementations communicate correctly, that is to say that messages sent by one implementation must be correct (this is done by the output verification) and actually received by the other implementation. The verification of this reception corresponds to input management. However only outputs can be observed by testers. Thus, verifying that an input μ is actually received implies to determine the set of outputs that can happen only if this reception is actually executed. This set of outputs is calculated based on causal dependencies. The set of outputs (*without quiescence*) on S that causally depend on the input μ after the trace σ is noted $CDep(S, \sigma, \mu)$ and defined by $CDep(S, \sigma, \mu) = \{\alpha_i \in \Sigma_O^S | \forall (q, q'), q_0^S \stackrel{\sigma}{\Rightarrow} q \xrightarrow{\mu} q', \exists q_i, q' \stackrel{\sigma_i.\alpha_i}{\Longrightarrow} q_i, \sigma_i \in (\Sigma^S \setminus \Sigma_O^S)^* \cup \{\epsilon\}\}$, where $\sigma_i \in (\Sigma^S \setminus \Sigma_O^S)^* \cup \{\epsilon\}$ is the path associated to the output $\alpha_i \in CDep(S, \sigma, \mu)$.
Based on causal dependency events, a condition for input verification can be defined. We give here the condition for the verification of the execution of an input $\bar{\mu}$ by I_2 (the corresponding output is μ sent by I_1). This condition considers each output μ executed by I_1 after a trace σ of the interaction. This trace can be decomposed into σ_1 (events of I_1) and σ_2 (events of I_2). The input management condition says that the reception of $\bar{\mu}$ by I_2 implies the observation of an output that causally depends on $\bar{\mu}$. Some events may be executed between μ and $\bar{\mu}$ (noted by trace $\sigma' \in ((\Sigma^{S_1} \cup \Sigma^{S_2}) \setminus \bar{\mu})^* \cup \{\epsilon\})$ and between $\bar{\mu}$ and the output that causally depends on $\bar{\mu}$ (trace $\sigma_i \in (\Sigma_I^{S_2})^* \cup \{\epsilon\}$). Formally, the condition is described by:
$\forall \sigma \in Traces(S_1 \|_{\mathcal{A}} S_2)$, $\sigma_1 = \sigma / \Sigma^{S_1} \in Traces(\Delta(S_1))$, $\sigma_2 = \sigma / \Sigma^{S_2} \in Traces(\Delta(S_2))$, $\forall \mu \in Out_{\Sigma_L^{I_1}}(\Delta(I_1), \sigma_1)$,
$\forall \sigma' \in ((\Sigma^{S_1} \cup \Sigma^{S_2}) \setminus \bar{\mu})^* \cup \{\epsilon\}$, $\sigma.\mu.\sigma'.\bar{\mu} \in Traces(S_1 \|_{\mathcal{A}} S_2)$,
$\bar{\mu} \in In(I_2, \sigma_2.(\sigma' / \Sigma^{I_2})) \Rightarrow$
$Out(I_2, \sigma_2.(\sigma' / \Sigma^{I_2}).\bar{\mu}.\sigma_i) \subseteq CDep(S_2, \sigma_2.(\sigma' / \Sigma^{I_2}), \bar{\mu})$ with $\sigma_i \in (\Sigma_I^{S_2})^* \cup \{\epsilon\}$.

3.3 Interoperability formal definitions

Even though some formal definitions exist in [5, 8], there is no precise characterization for interoperability (*iop* for short in the following). Here, we present some formal definitions, called *iop criteria*. They consider different possible architectures (see Section 2.1) for testing the interoperability of two IUTs.
The **unilateral iop criterion** iop_U (point of view of I_1) considers interfaces of IUT I_1 while interacting with I_2. It says that, after a trace of S_1 observed during the interaction, all outputs (and quiescence) observed in I_1 must be foreseen in S_1, and that I_1 must be able to receive outputs sent by I_2 via its lower interfaces.

Definition 2 (Unilateral iop criterion iop_U). $I_1 iop_U I_2 =$
$\forall \sigma_1 \in Traces(\Delta(S_1))$, $\forall \sigma \in Traces(S_1 \|_{\mathcal{A}} S_2)$,
$\sigma / \Sigma^{S_1} = \sigma_1 \Rightarrow Out((I_1 \|_{\mathcal{A}} I_2) / \Sigma^{S_1}, \sigma_1) \subseteq Out(\Delta(S_1), \sigma_1)$
and $\forall \sigma_1 = \sigma / \Sigma^{S_1} \in Traces(\Delta(S_1))$ *such that* $\sigma \in Traces(S_1 \|_{\mathcal{A}} S_2)$, $\forall \mu \in Out(I_2, \sigma / \Sigma^{I_2})$, $\forall \sigma' \in [(\Sigma^{S_1} \cup \Sigma^{S_2}) \setminus \bar{\mu}]^* \cup \{\epsilon\}$, $\sigma.\mu.\sigma'.\bar{\mu} \in Traces(S_1 \|_{\mathcal{A}} S_2)$, $\bar{\mu} \in In(I_1, \sigma_1.(\sigma' / \Sigma^{I_1})) \Rightarrow Out(I_1, \sigma_1.(\sigma' / \Sigma^{I_1}).\bar{\mu}.\sigma_i) \subseteq CDep(S_1, \sigma_1.(\sigma' / \Sigma^{I_1}), \bar{\mu})$, $\sigma_i \in (\Sigma_I^{S_1})^* \cup \{\epsilon\}$

The **bilateral total iop criterion** iop_B is verified iff both (on I_1 point of view and I_2 point of view) unilateral criteria are verified: $I_1\ iop_B\ I_2$ ($= I_2\ iop_B\ I_1$) $= I_1\ iop_U\ I_2 \wedge I_2\ iop_U\ I_1$.

The **global iop criterion** considers both kinds of interfaces and both IUTs globally. It says that, after a trace of the specification interaction, all outputs (and quiescence) observed during the interaction of the implementations must be foreseen in the specifications, and that outputs sent by one IUT via its lower interfaces must be effectively received by the interacted IUT. Contrary to iop_U and iop_B that are used in specific contexts where some interfaces are not accessible, this iop criterion iop_G corresponds to the most used testing architecture.

Definition 3 (Global iop criterion iop_G). $I_1 iop_G I_2 =$
$\forall \sigma \in Traces(S_1 \|_{\mathcal{A}} S_2)$, $Out(I_1 \|_{\mathcal{A}} I_2, \sigma) \subseteq Out(S_1 \|_{\mathcal{A}} S_2, \sigma)$
and $\forall \{i,j\} = \{1,2\}$, $i \neq j$,
$\forall \sigma \in Traces(S_i \|_{\mathcal{A}} S_j)$, $\sigma_i = \sigma / \Sigma^{S_i} \in Traces(S_i)$, $\sigma_j = \sigma / \Sigma^{S_j} \in Traces(S_j)$, $\forall \mu \in Out(I_i, \sigma / \Sigma^{S_i})$, $\forall \sigma' \in [(\Sigma^{S_i} \cup \Sigma^{S_j} \cup \{\delta(i), \delta(j)\}) \setminus \bar{\mu}]^* \cup \{\epsilon\}$, $\sigma.\mu.\sigma'.\bar{\mu} \in Traces(S_i \|_{\mathcal{A}} S_j)$, $\bar{\mu} \in In(I_j, \sigma_j.(\sigma' / \Sigma^{I_j})) \Rightarrow Out(I_j, \sigma_j.(\sigma' / \Sigma^{I_j}).\bar{\mu}.\sigma_k) \subseteq CDep($ S_j, $\sigma_j.(\sigma' / \Sigma^{I_j})$, $\bar{\mu})$, $\sigma_k \in (\Sigma_I^{S_j})^* \cup \{\epsilon\}$

In [5], we prove the equivalence of the global criterion with the so-called bilateral iop criterion iop_B in terms of non-interoperability detection. However, the iop criteria defined in [5] only consider the output verification, that is the first part of the definition of iop_B and iop_G of this study. These latter criteria are still equivalent in terms of non-interoperability detection. Indeed, the causal-dependency based condition is defined with a bilateral point of view in both criteria. In next Section, we focus in the way to use this equivalence for developing methods to derive automatically iop test cases.

4 Interoperability test generation methods

In this section, we investigate the way to generate interoperability (iop for short in the following) tests based on the defined iop criteria. Applications of these methods are described in Section 5.

4.1 Test purposes, test cases and verdicts

Iop test purpose In practice, interoperability test case derivation is done based on test purposes. These test purposes are used by testers to describe the properties they want to test. An iop test purpose is an informal description of behaviors to be tested, in general an incomplete sequence of actions. Formally, a test purpose TP can be represented by a deterministic and complete IOLTS equipped with trap states used to select targeted behaviors. Complete means that each state allows all actions. In this study, we consider simplified iop test purposes with only one possible action after each state ($\forall\ \sigma,\ |\Gamma(TP, \sigma)| \leq 1$) and one

$Accept^{TP}$ trap state used to select the targeted behavior.
Iop test cases During interoperability tests, three kinds of events are possible: sending of stimuli to the upper interfaces of the IUTs, reception of inputs from these interfaces, and observation of events (input and output) on the lower interfaces. Thus, an iop test case TC can be represented by $TC = (Q^{TC}, \Sigma^{TC}, \Delta^{TC}, q_0^{TC})$, an extended version of IOLTS. $\{PASS, FAIL, INC\} \subseteq Q^{TC}$ are trap states representing interoperability verdicts. q_0^{TC} is the initial state. $\Sigma^{TC} \subseteq \{\mu|\bar{\mu} \in \Sigma_U^{S_1} \cup \Sigma_U^{S_2}\} \cup \{?(\mu)|\mu \in \Sigma_L^{S_1} \cup \Sigma_L^{S_2}\}$. $?(\mu)$ denotes the observation of the message μ on a lower interface. Notice that in interoperability testing μ can be either an input or an output. Δ^{TC} is the transition function.
Iop verdicts The execution of the iop test case TC on the system composed of the two IUTs gives an iop verdict: PASS, FAIL or INC. The meanings of the possible iop verdicts are PASS: no interoperability error was detected during the tests, FAIL: the iop criterion is not verified and INC (for Inconclusive): the behavior of the SUT seems valid but it is not the purpose of the test case.

4.2 Global interoperability test generation method

The global interoperability test generation method is based on the first part of the global iop criterion iop_G. This part focuses on the comparison between outputs (and quiescence) observed during the interaction of the implementations and outputs (and quiescence) foreseen in the specifications in the same situation. This method corresponds to what is done practically when writing iop test cases "by hand". It also corresponds to most approaches for automatic interoperability test generation (as in [7, 8, 14, 15, 16, 17]) even if these methods generally do not compute the complete specification interaction graph. This is why we also call it classical interoperability test generation method.
The global interoperability test generation method (see Figure 2(a)) begins with the construction of the asynchronous interaction $S_1 \| _{\mathcal{A}} S_2$ to have a model of the global system specification. Then $S_1 \| _{\mathcal{A}} S_2$ is composed with the test purpose TP. During this operation, two main results are calculated. First TP is validated. If the events composing TP are not found in the specifications (or not in the order described in TP), TP is not a valid Test Purpose. The composition is also used to keep only the events concerned by the Test Purpose (in the interaction of the two specifications). It calculates the different ways to observe/execute TP on the System Under Test (SUT) composed of the two IUTs.
A conformance test tool takes as entries a test purpose and a specification and computes the paths of the specification executing the test purpose. Thus, such a tool can be used for computing the paths executing TP in the specification interaction. In this case, the tool entries are TP and $S_1 \|_{\mathcal{A}} S_2$. However, some modifications needs to be applied to the test case obtained with this tool to derive interoperability test cases. Events on lower interfaces are not controllable in the interoperability context, contrary to the case of conformance testing.

One problem with this method (classical method) is that we can have state space explosion when calculating the interaction of the specifications [6]. Indeed,

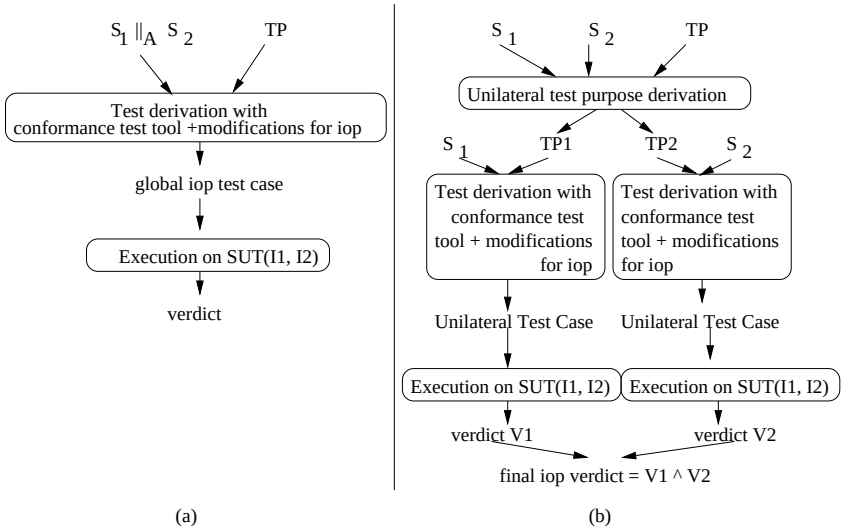

Fig. 2. Interoperability test generation: global and bilateral approaches

the state number of the specification asynchronous interaction is in the order of $O((n.m^f)^2)$ where n is the state number of the specifications, f the size of the input FIFO queue on lower interfaces and m the number of messages in the alphabet of possible inputs on lower interfaces. This result can be infinite if the size of the input FIFO queues is not bound. However, the equivalence -in terms of non-interoperability detection- between global and bilateral iop criteria (cf. theorem 1 in [5]) suggests that iop tests derived based on the bilateral iop criterion will detect the same non-interoperability situations as tests generated using the global interoperability test generation method.

4.3 Bilateral interoperability test generation method

The bilateral interoperability test generation method (see Figure 2(b)) is based on the first part of the iop_B criterion. This part focuses on the comparison between outputs (and quiescence) observed on the interfaces of the interacting implementations -each test case considering one IUT during the interaction- and outputs (and quiescence) foreseen in the corresponding specification.

Unilateral iop test purpose derivation The first step of the bilateral interoperability test generation method derives automatically two unilateral iop test purposes TP_{S_i} from the global iop test purpose TP. The algorithm of figure 3 shows how to derive these two unilateral iop test purposes. Let us consider an event μ of the iop test purpose TP and the construction of TP_{S_i}. If μ is an event of the specification S_i, μ is added to the unilateral iop test purpose TP_{S_i}. If μ is an event from the specification S_j, there are two possibilities. Either the

event is to be executed on lower interfaces: in this case, the mirror event $\bar{\mu}$ is added to TP_{S_i}; or, the event is to be executed on the upper interfaces: in this case, the algorithm searches a predecessor of μ, such that this predecessor is an event to be executed on lower interfaces. The algorithm adds the mirror of this predecessor to the unilateral iop test purpose TP_{S_i}.

Input: S_1, S_2: specification, TP: iop test purpose
Output: $\{TP_{S_i}\}_{i=1,2}$;
Invariant: $S_k = S_{3-i}$ (* S_k is the other specification *); $TP = \mu_1 ... \mu_n$
Initialization: $TP_{S_i} := \epsilon \; \forall i \in \{1, 2\}$;
for ($j = 0$;$j \leq n$;j++) **do**
  if ($\mu_j \in \Sigma_L^{S_i}$) **then** $TP_{S_i} := TP_{S_i}.\mu_j$; $TP_{S_k} := TP_{S_k}.\bar{\mu_j}$ **end(if)**
  if ($\mu_j \in \Sigma_L^{S_k}$) **then** $TP_{S_i} := TP_{S_i}.\bar{\mu_j}$; $TP_{S_k} := TP_{S_k}.\mu_j$ **end(if)**
  if ($\mu_j \in \Sigma_U^{S_i}$) **then** $TP_{S_i} := TP_{S_i}.\mu_j$; TP_{S_k} :=add_precursor(μ_j, S_i, TP_{S_k}) **end(if)**
  if ($\mu_j \in \Sigma_U^{S_k}$) **then** $TP_{S_k} := TP_{S_k}.\mu_j$; TP_{S_i} :=add_precursor(μ_j, S_k, TP_{S_i}) **end(if)**
  if ($\mu_j \notin \Sigma^{S_k} \cup \Sigma^{S_i}$) **then** error(TP not valid: $\mu_j \notin \Sigma^{S_1} \cup \Sigma^{S_2}$) **end(if)**
end(for)

function add_precursor(μ, S, TP): **return** TP
  $\sigma_1 := TP$; a_j =last_event(σ_1)
  while $a_j \in \Sigma_U^S$ **do** σ_1=remove_last(σ_1); a_j =last_event(σ_1) **end(while)**
  $M = \{q \in Q^S$; $\exists$ q'|(q, $\bar{a}_j, q') \wedge \sigma = \bar{a}_j.\omega.\mu \in Traces(q)\}$
  if ($\forall q \in M$, $\sigma \notin Traces(q)$) **then** error(no path to μ) **end(if)**
  while (e=last_event(ω) $\notin \Sigma_L^S \cup \{\epsilon\}$) **do** ω=remove_last(ω) **end(while)**
  if (e $\in \Sigma_L^S$) **then** $TP_S = TP_{S_i}.\bar{e}$ **end(if)**

Fig. 3. Algorithm to derive TP_{S_i} from TP

Some additional functions are used in the algorithm of figure 3. Let us consider a trace σ and an event a. The function *remove_last* is defined by : remove_last($\sigma.a$) = σ. The function *last_event* is defined by : last_event(σ)= ϵ if $\sigma = \epsilon$ and last_event(σ)= a if $\sigma = \sigma_1.a$. The *error* function returns the cause of the error and exits the algorithm.

Unilateral iop test case generation The second step of the bilateral interoperability test generation method is the generation of two unilateral test cases from the unilateral test purposes and the specifications. The same test generation algorithm is executed for TP_{S_1} with S_1 and for TP_{S_2} with S_2. This algorithm calculates on-the-fly the interaction between the unilateral iop test purpose and the corresponding specification to find in the specification the paths executing the test purpose. This step can be done by using a conformance test generation tool (for example TGV).
However, as lower interfaces are not controllable in interoperability testing (contrary to conformance testing), controllable conformance test cases can not always be reused for interoperability. Indeed, a test case is controllable if the tester does not need to choose arbitrarily between different events. In conformance, inputs on lower interfaces correspond to outputs of the tester: a controllable conformance test case only considers one of the possible inputs on lower interfaces.

In interoperability testing, inputs on lower interfaces are sent by the other implementation. An interoperability test case must take into account *all* possible inputs on lower interfaces. The complete test graph is an IOLTS which contains all sequences corresponding to a test purpose: all the inputs of the implementation that correspond to the test purpose are considered. Thus, to have test cases usable in interoperability context, the conformance tool used in this step for interoperability test generation must compute the complete test graph.
Moreover, some modifications are needed on the test cases TC'_1 and TC'_2 generated by the conformance test tool to obtain the unilateral iop test cases TC_1 and TC_2 that will be executed unilaterally on the corresponding IUT in the SUT. These modifications are needed because lower interfaces are only observed (not controlled) in interoperability context. For example, if an event $l!m$ exists in the test case obtained from TGV (which means that the tester connected to interface l must send the message m to the lower interface of the IUT), this will correspond to $?(l?m)$ in the interoperability test case. This means that the interoperability tester observes that a message m is received on the interface l. No changes are made on the test cases for events on the upper interfaces as these interfaces are observable and controllable: a message can be sent (and received) by the tester to the IUT on these interfaces.
The execution of both interoperability test cases return two verdicts. The "bilateral" verdict is obtained by combining these local verdicts with the following obvious rules: $PASS \wedge PASS = PASS$, $PASS \wedge INC = INC$, $INC \wedge INC = INC$, and $FAIL \wedge (FAIL \vee INC \vee PASS) = FAIL$.

Complexity The first step of the bilateral interoperability test generation method is linear in the maximum size of specifications. Indeed, it is a simple path search algorithm. The second step is also linear in complexity, at least when using TGV [19]. Thus, it costs less than the calculation of $S_1 \|_{\mathcal{A}} S_2$ needed in the global interoperability test generation method. Moreover the bilateral interoperability test generation method can be used to generate iop test cases in situations where generating test cases with the global interoperability test generation method is impossible due to state-space explosion problem.

4.4 Causal dependency based algorithm completing both methods

One objective of interoperability is to verify the communication between the IUTs. Moreover, iop test purposes may end with an input. This latter situation occurs, for example, in the unilateral test purposes derived by bilateral method. For example, if the iop test purpose ends with an output on the lower interface, its mirror event (an input) is added -as last event- to one of the derived test purpose. In this case, the conformance test tool (TGV) generates a test case without postamble: the last event of the test case is the input given as objective by the test purpose. However, this input is not observable. An algorithm based on input-output causal dependencies is used to know if this input was actually executed. It completes iop test cases obtained by bilateral method (or test cases generated by classical method based on an iop test purpose ending with an input)

by producing outputs that help in verifying that the input is actually executed. Thus, the algorithm based on causal dependencies completes and refines iop test cases generated by bilateral (or global) method. It takes as entry the iop test case to complete: the last event of this test case is the input μ. It returns the outputs that are causally dependent of this input μ. For computing the set of causal dependency events (associated with the paths to these events), this algorithm, see Figure 4, considers each event of the set $\Gamma(S_1, \sigma.\mu)$ to find an output in each trace following the considered input. The obtained outputs are used to verify the actual reception of the input μ and thus, to complete test cases based on the iop test purpose containing this input.

Input: S_1: Specification, σ: Trace of event (after projection on S_1), μ: input of S_1
Output: $CDep(S_1, \sigma, \mu)$, $\{\sigma'\}$: set of traces between μ and an output, m: number of events in $CDep$
Initialization: $\Gamma := \Gamma(S_1, \sigma.\mu)$; $m :=$nb_event(Γ)
for $(i := 0; i \leq m; i++)$ **do**
  create(find[i]); find[i]=false; create($\sigma'[i]$); $\sigma'[i] := \epsilon$ **end(for)**
BEGIN
while $\exists x(x < n)$, find[x]:=false **do**
  for $(i := 0; i < m; i++)$ **do**
    if (find[i]=false) **do** $Evt := \Gamma(S_1, \sigma.\mu.\sigma'[i])$
      if $(Evt(0) \in \Sigma_O^{S_1})$ **do** find[i]:=true; Add(Evt(0), $CDep(S_1, \sigma, \mu)$)
      else $\sigma'[i] := \sigma'[i]$.Evt(0) **end(if)**
      if (nb_event(Evt)> 1) **do**
        for$(j := 1;$ j $\leq$ nb_event(Evt); $j++)$ **do**
          m++; create($\sigma'[m]$); $\sigma'[m] := \sigma'[i]$
          create(find[m]); find[m]:=false
          if $(Evt(j) \in \Sigma_O^{S_1})$ **do** find[m]:=true; Add(Evt(j), $CDep(S_1, \sigma, \mu)$)
          else $\sigma'[m] := \sigma'[m]$.Evt(j) **end(if)**
        end(for)
      end(if)
    end(if)
  end(for)
end(while)
END

Fig. 4. Exploration of S_1 to find causally dependent outputs of input μ

4.5 Implementation of iop test generation methods

In [5], we show the equivalence that allows the definition of the bilateral algorithm. However, the definitions and methods were not complete as inputs were not verified (there was no condition and no algorithm based on causal-dependencies) and the algorithms presented were not tested practically.
The methods presented in this Section were implemented using the CADP toolbox [20]. The conformance test generation tool TGV (Test Generation using Verification techniques) [3] is integrated in this toolbox which also contains an API for manipulating IOLTS. These methods were applied to the generation of iop test cases for a connection protocol. It is described in next Section.

5 Application on a connection protocol

Figure 5 describes three specifications for a connection protocol. S_1 and S_2 are respectively the specifications of the client and server. $U1?CNR$ is a connection request from the upper layer, $l1!cnr$ (resp. $l2?cnr$) the request sent (resp. received) to the peer entity, $l2!ack/l2!nack$ the positive or negative response, and $U1!ACK/U1!NACK$ the response forwarded to the upper layer. The specification S represents both client and server parts.

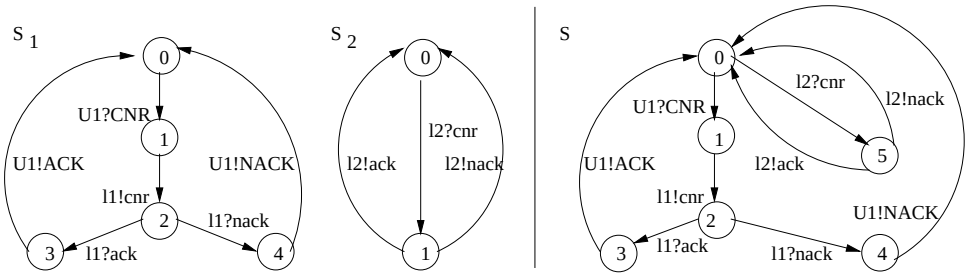

Fig. 5. Examples of specifications: S_1, S_2 and S

5.1 A client/server example

Let us consider the three iop test purposes of figure 6(a). These iop test purposes are applicable for the System Under Test (SUT) composed of two IUTs implementing respectively S_1 and S_2. For example, TP_1 means that, after the reception by I_1 (implementing S_1) of a connection demand on its upper interface $U1$, this IUT must send a connection acknowledgment on its upper interface $U1$.

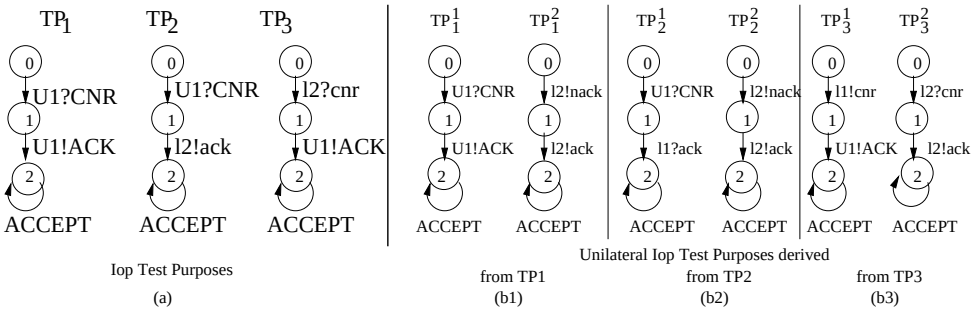

Fig. 6. Iop test purpose TP_1, TP_2 and TP_3, and derived Unilateral Test Purposes

In figure 6 (b1), TP_1^1 and TP_1^2 are the unilateral test purposes derived using the algorithm of figure 3 for TP_1 and respectively specifications S_1 and S_2. In the

same way, TP_2^1 and TP_2^2 of figure 6 (b2) (resp. TP_3^1 and TP_3^2 of figure 6 (b3)) are derived from TP_2 (resp. TP_3). The same notation will be used for test cases in the following.

When deriving the unilateral iop test purposes, for events on lower interfaces, the returned event is either the event itself, or its mirror. For event $U1!ACK$, as its predecessor is $\mu = l1?ack$, the returned event is $\bar{\mu} = l2!ack$ (TP_1^2 and TP_3^2) or $U1!ACK$ (TP_1^1 and TP_3^1). The difficulty is for deriving an event from $U1?CNR$ for TP_1^2 and TP_2^2. In S_1, this event is the first possible event after the initial state. Its predecessor must be found in the paths that bring back the entity in its initial state after some execution. The first predecessor found is $U1!NACK$. As this event is not an event of the interaction, the algorithm continues one more step to find $l1?nack$ as predecessor, and then returns its mirror $l2!nack$.

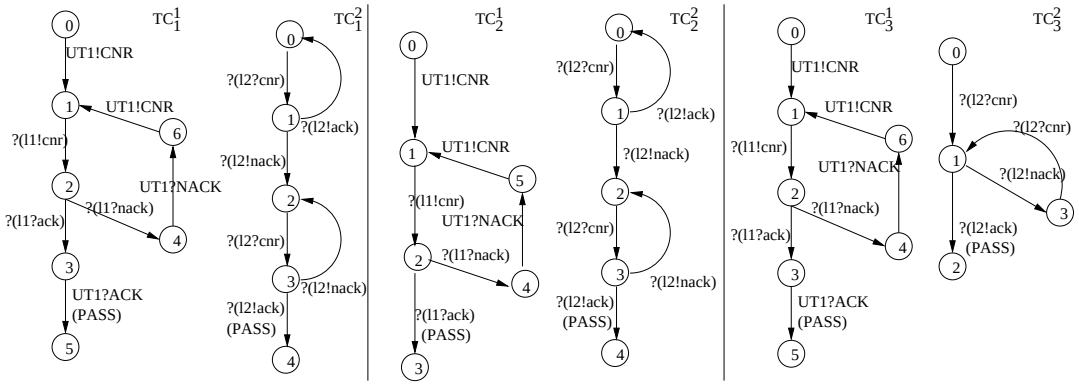

Fig. 7. Test Cases by bilateral method for specifications S_1 and S_2

The second step of the bilateral iop test generation method corresponds to the use of TGV conformance test tool on a unilateral test purpose and the corresponding specification. Figure 7 gives the test cases obtained for the test purposes of figure 6. The results on Figure 7 gives the test cases modified for interoperability. $UT1$ is the tester connected to upper interface $U1$.

Now, let us see what happens when using the classical approach for iop test generation to compare test cases generated by both methods. The first step of the classical method is the calculation of the specification interaction. Then, we can use TGV to generate test cases for test purposes of Figure 6. The obtained global iop test cases are in Figure 8. We can remark that, for the three situations (comparing traces in Figures 7 and 8), the same execution paths lead to the same verdicts. Thus, the iop test cases generated with both methods are equivalent in terms of verdicts.

For TP_2, we can remark that $TC_2^1 \|_{\mathcal{A}} TC_2^2$ ends with an input ($l1?ack$) that is not in TC_2 (excluding this event, TC_2 and $TC_2^1 \|_{\mathcal{A}} TC_2^2$ contain the same traces). This is due to the test purpose derivation (cf. Section 4.4): the unilateral test pur-

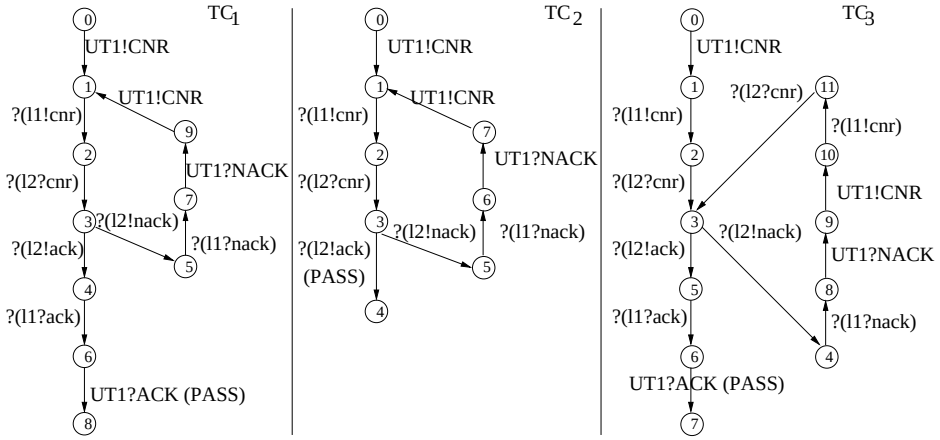

Fig. 8. Test Cases from TGV for the interaction of S_1 and S_2

pose generated for S_1 ends with an input. To complete this iop test case (TC_2^1), we can either add a postamble returning to the initial state, either use the causal dependency based algorithm. In this simple example (specification S_1), only the event $U1!ACK$ will be added with causal dependency event method.
To summarize, the application of both method on this connection protocol confirms the equivalence in terms of verdicts. Even though the generated iop test cases are not the same, the execution of the same traces leads to the same verdicts. Thus, the same non-interoperability situation are detected with both our method and the classical method.

5.2 Specification describing both entities as client and server

Both methods were also applied on the specification S (Figure 5) describing both client and server parts (same test purposes). The interaction $S\|_{\mathcal{A}}S$, calculated for classical approach, is composed of 454 states and 1026 transitions with input queues of each system bounded to one message. The following table gives the number of states s and transitions t (noted s/t in the table) of results derived considering a queue size of 3. Line 1 and 2 give the state and transition numbers for unilateral test cases derived by bilateral method considering S as specification for both systems (lines 1 and 2: S as specification respectively for systems 1 and 2). The third line gives numbers for the interaction of these unilateral test cases TC^1 and TC^2. Finally line 4 gives results for global methods. With a queue size of 3,the specification interaction has 47546 states and 114158 transitions.

	TP_1	TP_2	TP_3
Unilateral iop test case TC^1	9/17	8/16	9/17
Unilateral iop test case TC^2	13/24	13/24	12/22
$TC^1\|_{\mathcal{A}}TC^2$	19546/57746	19468/57614	19405/57386
Global test case TC	54435/120400	18014/40793	54456/120443

We observe that we can derive iop test cases considering a queue size of 3 via both classical and bilateral methods. However, due to the difference in state and transition number between both methods, unilateral test cases obtained by bilateral method are more usable, for example for controlling the execution of the interoperability test cases. Moreover, state space explosion problem can occur when using the global method: results in the previous table are finite only because we consider bounded FIFO queues. We were not able to compute $S\|_{\mathcal{A}}S$ for a queue size limited to 4 places. But the bilateral method gives iop test cases with the same state and transition numbers as in the previous table. This shows that the bilateral method can be used to generate iop test cases even for specifications that produce state space explosion problem. Moreover, these test cases are not dependent of the queue size.

6 Conclusion

In this paper, we present interoperability formal definitions that deal with both purposes of interoperability: implementations must interact correctly and the expected service must be provided. A new interoperability test generation method is proposed based on these formal definitions. This method has been implemented using the CADP toolbox. It avoids the well-known state-space explosion problem that occurs when using classical methods. This is the important result of our study and it is confirmed by the application on a connection protocol. Moreover, we show that the so-called bilateral interoperability test derivation method allows us to generate interoperability test cases in situations where it would have been impossible with the traditional method because of state space explosion problem.
As future work, we will study the generalization of the formal interoperability definitions and test generation methods to the much more complex context of more than two implementations. We will also study how to apply the described method to a distributed testing architecture.

References

[1] ISO. Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Parts 1-7. *International Standard ISO/IEC 9646/1-7*, 1992.

[2] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 46–65. Springer-Verlag, 1999.

[3] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[4] J. Tretmans and E. Brinksma. Torx: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the First European Conference on Model-Driven Software Engineering*, Nurnberg, Germany, December 2003.

[5] A. Desmoulin and C. Viho. Formalizing interoperability for test case generation purpose. In *IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, Columbia, MD, USA, September 2005.

[6] R. Castanet and O. Koné. Deriving coordinated testers for interoperability. In O. Rafiq, editor, *Protocol Test Systems*, volume VI C-19, pages 331–345, Pau-France, 1994. IFIP, Elsevier Science B.V.

[7] Soonuk Seol, Myungchul Kim, Sungwon Kang, and Jiwon Ryu. Fully automated interoperability test suite derivation for communication protocols. *Comput. Networks*, 43(6):735–759, 2003.

[8] R. Castanet and O. Kone. Test generation for interworking systems. *Computer Communications*, 23:642–652, 2000.

[9] T. Walter, I. Schieferdecker, and J. Grabowski. Test architectures for distributed systems : state of the art and beyond. In Petrenko and Yevtushenko, editors, *Testing of Communicating Systems*, pages 149–174. IFIP, Kap, September 1998.

[10] Sébastien Barbin, Lénaïck Tanguy, and César Viho. Towards a formal framework for interoperability testing. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, pages 53–68, Cheju Island, Korea, Août 2001.

[11] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In J. Wu, S. Chanson, and Q. Gao, editors, *Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China*, pages 25–40. Kluwer Academic Publishers, October 1999.

[12] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In G.V. Bochman, R. Dssouli, and A. Das, editors, *Fifth inteernational workshop on protocol test systems*, pages 55–66, North-Holland, 1993. IFIP Transactions.

[13] A. Desmoulin and C. Viho. Quiescence Management Improves Interoperability Testing. In *17th IFIP International Conference on Testing of Communicating Systems (Testcom)*, Montreal, Canada, May-June 2005.

[14] Khaled El-Fakih, Vadim Trenkaev, Natalia Spitsyna, and Nina Yevtushenko. Fsm based interoperability testing methods for multi stimuli model. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2004.

[15] Nancy D. Griffeth, Ruibing Hao, David Lee, and Rakesh K. Sinha. Integrated system interoperability testing with applications to voip. In *FORTE/PSTV 2000: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*. Kluwer, B.V., 2000.

[16] G. Bochmann, R. Dssouli, and J. Zhao. Trace analysis for conformance and arbitration testing. *IEEE transaction on software engeneering*, 15(11):1347–1356, November 1989.

[17] J. Gadre, C. Rohrer, C. Summers, and S. Symington. A COS study of OSI interoperability. *Computer standards and interfaces*, 9(3):217–237, 1990.

[18] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2001. ACM Press.

[19] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV '96: 8th International Conference on Computer Aided Verification*, London, UK, 1996.

[20] H. Garavel, F. Lang, and R. Mateescu. An overview of cadp 2001. Technical Report 0254, INRIA, 2001.