

# A Tagging Protocol for Asynchronous Testing

Puneet Bhateja

IRISA / INRIA Rennes

Campus Universitaire de Beaulieu

35042 Rennes Cedex - FRANCE

puneet.bhateja@inria.fr

**Abstract**—Conformance testing has a rich underlying theory popularly called IOCO-test theory. In the realm of IOCO-test theory, this paper addresses the issue of testing a component of an *asynchronously communicating distributed system*. Testing a system which communicates asynchronously (i.e., through some medium) with its environment is more difficult than testing a system which communicates synchronously (i.e., directly without any medium). What impedes asynchronous testing is that the actual behavior of the implementation under test (IUT) appears distorted and infinite to the tester. This impediment consequently renders the problem of generating a complete test suite, from the given specification of the IUT, infeasible. To this end, this paper contributes by proposing a tagging protocol which when implemented by the asynchronously communicating distributed system will make the problem of generating a complete test suite, from the specification of any of its component, feasible. Further, this paper describes how to generate the test suite from the given specification of the component.

**Keywords**-Synchronous testing; asynchronous testing; queue context; tagging protocol.

## I. INTRODUCTION

Conformance testing is an operational way to check the functional behavior of an IUT vis-a-vis its specification. Jan Tretmans propounded the IOCO-test theory [7] to explain the underlying principles of conformance testing. A characteristic feature of this theory is that the specification, the IUT, the test case are all modeled by an input output labeled transition system (IOLTS), and the conformance is modeled by a binary relation over the set of all IOLTSs. The execution of a test case against the IUT is described by communication between the IOLTS depicting the test case and the IOLTS depicting the IUT. In synchronous testing, the IOLTSs corresponding to the IUT and the test case communicate with each other by doing common actions simultaneously, whereas in asynchronous testing they communicate through a pair of first-in-first-out (FIFO) queues called queue context.

Over the years, IOCO-test theory has given impetus to many popular test generation tools such as TorX [6], TGV [2], etc. However, all these tools work on the underlying principle of synchronous testing, whereas there are not many test generation techniques in the context of asynchronous testing. One reason that hinders development in this direction

is that the asynchronous behavior (the behavior visible through the queue context) of the IUT is always infinite and distorted even if the synchronous behavior (actual behavior) of the IUT is finite. What is more, asynchronous conformance bears no relation with synchronous conformance. In other words, it could be the case that the asynchronous behavior of the IUT is in conformance with the asynchronous behavior of the specification (the behavior of the specification observed presumably through a similar queue context), whereas the actual behavior of the IUT is not in conformance with the actual behavior of the specification, and vice-versa.

This paper proposes a tagging protocol whereby each component of the asynchronously communicating distributed system tags each message with “0” or “1”, before sending it to its peer component. The tag does not incur much overhead, as it occupies just one bit. However, even with this 1-bit tagging protocol, we are able to deduce the following relation: The synchronous behavior of the IUT is in conformance with the synchronous behavior of the specification if and only if the tagged asynchronous behavior of the IUT is a subset of the tagged asynchronous behavior of the specification. Based on the protocol, this paper describes a methodology to generate test cases which constitute a complete test suite with respect to the given specification.

We now give a brief account of the related work and the state of the art. First of all, Verhaard et al. in [9] showed that the test cases for synchronous testing cannot be used for asynchronous testing. They showed what a typical test case for asynchronous testing would be with respect to a set of execution sequences in the asynchronous behavior of the specification.

In [3] Jard et al. proposed a time-stamping scheme which time stamps every output generated by the IUT. As a result, the asynchronous time-stamped behavior of the IUT is in conformance with the asynchronous time-stamped behavior of the specification if and only if the actual behavior of the IUT is in conformance with the actual behavior of the specification. However, a serious drawback of their approach is that the size of the time stamp is directly proportional to the length of the computation. Thus, their scheme makes

sense for only small length computations. Contrary to this, our tagging scheme is of constant length.

Recently, Weiglhofer et al. [10] showed that if an IUT in any state can either exclusively perform input actions or output actions, then it is possible to generate a complete test suite which would test the IUT for the relation *ioco* through a pair of FIFO queues. In this scheme, the tester and the IUT synchronize with each other in the sense that at any time exactly one of them outputs and the other inputs. Contrary to this, our approach neither holds any assumption about the structure of IUT nor assumes any synchrony between the tester and the IUT.

This paper is organized into six sections as follows: Section 1 comprises the ongoing introduction; Section 2 defines the IOLTS model and some notations relevant to it; Section 3 explains synchronous testing in the context of IOCO-test theory; Section 4 explains asynchronous testing in the context of IOCO-test theory; Section 5 explains the tagging protocol and its consequences; Section 6 describes the methodology of generating test cases from the given specification; and finally Section 7 makes some concluding remarks.

## II. IOLTS: DEFINITIONS AND NOTATIONS

An IOLTS (input output labeled transition system) is a state-based model which is widely used to explain the observed behavior of an interactive system [4], [8]. IOLTS abstracts away the actions that are internal to the system, while it exhibits in a mutually distinctive manner the actions whereby the system sends/receives messages to/from its environment. Formally, an IOLTS can be defined as follows:

**Definition 1:** An IOLTS is a quadruple  $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ , where  $Q^M$  is a non empty set of states;  $A^M$  is a set of external (visible) actions, and is further partitioned into an input alphabet  $A_I^M$  and an output alphabet  $A_O^M$ ;  $q_0^M \in Q^M$  is the initial state of the IOLTS; and  $\rightarrow_M \subseteq Q^M \times (A^M \cup \{\tau\}) \times Q^M$  is a transition relation, where  $\tau \notin A^M$  is some internal (invisible) action.

We now state some definitions and notations with respect to an IOLTS  $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ .

- $\forall q, q' \in Q^M, \forall a \in A^M : (q \xrightarrow{a}_M q')$  is true iff  $(q, a, q') \in \rightarrow_M$ .  $\forall q \in Q^M, \forall a \in A^M : (q \xrightarrow{a}_M)$  is true iff  $\exists q' \in Q^M : (q, a, q') \in \rightarrow_M$ . We can generalize this to strings of all lengths.  $\forall q, q' \in Q^M, \forall a_1.a_2.\dots.a_n \in (A^M)^* : q \xrightarrow{a_1.a_2.\dots.a_n}_M q'$  is true iff  $\exists q_1, q_2, \dots, q_{n-1} : (q \xrightarrow{a_1}_M q_1) \wedge (q_1 \xrightarrow{a_2}_M q_2) \wedge \dots \wedge (q_{n-1} \xrightarrow{a_n}_M q')$ .
- If  $M$  changes its state from  $q_1$  to  $q_2$  without performing any visible action, we denote it by  $q_1 \xrightarrow{\tau}_M q_2$ . Formally,  $\forall q_1, q_2 \in Q^M : q_1 \xrightarrow{\tau}_M q_2$

iff  $(q_1 = q_2) \vee (q_1 \xrightarrow{\tau.\tau.\dots.\tau}_M q_2)$ .  $\forall X \subseteq Q^M : (X \text{ after } \epsilon)_M = \{q_2 | \exists q_1 \in X : q_1 \xrightarrow{\tau}_M q_2\}$ .

- For  $\sigma \in (A^M)^*$  and  $X \subseteq Q^M, \sigma \downarrow_X$  is called the projection of  $\sigma$  on  $X$ . It can be defined inductively. Base case:  $\epsilon \downarrow_X = \epsilon$ . Induction step:  $\sigma \downarrow_X = a.(\sigma' \downarrow_X)$ , when  $a \in X$  and  $\sigma = a.\sigma'$ ;  $\sigma \downarrow_X = \sigma' \downarrow_X$ , when  $a \notin X$  and  $\sigma = a.\sigma'$
- If  $M$  changes its state from  $q_1$  to  $q_2$  while performing a sequence  $\sigma \in (A^M)^*$  of visible actions, we denote it by  $q_1 \xrightarrow{\sigma}_M q_2$ . Formally, this can be defined inductively as follows. Base case: we define  $\forall q, q' \in Q^M, \forall a \in A^M : q \xrightarrow{a}_M q' \equiv \exists q_1, q_2 \in Q^M : (q \xrightarrow{\tau}_M q_1) \wedge (q_1 \xrightarrow{a}_M q_2) \wedge (q_2 \xrightarrow{\tau}_M q')$ . Induction step:  $\forall q, q' \in Q^M, \forall \sigma \in (A^M)^*, \forall a \in A^M : q \xrightarrow{\sigma.a}_M q' \equiv \exists q_1 \in Q^M : (q \xrightarrow{\sigma}_M q_1) \wedge (q_1 \xrightarrow{a}_M q')$ . Further,  $\forall q \in Q^M, \forall \sigma \in (A^M)^* : q \xrightarrow{\sigma}_M$  is true iff  $\exists q' \in Q^M : q \xrightarrow{\sigma}_M q'$ . Also,  $\forall q, q' \in Q^M : q \xrightarrow{\tau}_M q'$  iff  $\exists \sigma \in (A^M)^* : q \xrightarrow{\sigma}_M q'$ .
- The extensional behavior of an IOLTS  $M$  can be described by the set  $Traces(M) = \{\sigma \in (A^M)^* | q_0^M \xrightarrow{\sigma}_M\}$ . Intuitively,  $Traces(M)$  represents the set of traces that  $M$  can potentially execute.
- An IOLTS  $M$  is deterministic when  $\forall q, q_1, q_2 \in Q^M, \forall \sigma \in (A^M)^* : (q \xrightarrow{\sigma}_M q_1) \wedge (q \xrightarrow{\sigma}_M q_2)$  implies  $(q_1 = q_2)$ . Informally,  $M$  is deterministic, if (1) at every state in  $M$ , there is no choice between transitions on the same action, and (2) there is no transition in  $M$  on an invisible action  $\tau$ . Due to non determinism, the control of an IOLTS  $M$  can reach different states after executing a trace. We define  $\forall q \in Q^M, \forall \sigma \in (A^M)^* : (q \text{ after } \sigma) = \{q' \in Q^M | q \xrightarrow{\sigma}_M q'\}$ . This definition can be extended to sets of states also. We define  $\forall X \subseteq Q^M : (X \text{ after } \sigma) \equiv \bigcup_{q \in X} (q \text{ after } \sigma)$
- $\forall q \in Q^M : In(q) = \{a \in A_I^M | q \xrightarrow{a}_M\}$  is the set of input actions enabled in  $q$ .  $M$  is input complete if and only if  $\forall q \in Q^M : In(q) = A_I^M$ .  $\forall q \in Q^M : Out(q) = \{x \in A_O^M | q \xrightarrow{x}_M\}$  is the set of output actions enabled in  $q$ . The set of output actions enabled in a set of states  $P \subseteq Q^M$  is denoted by  $Out(P) = \bigcup_{q \in P} Out(q)$ .
- Let  $M_1 = (Q^{M_1}, A^{M_1}, \rightarrow_{M_1}, q_0^{M_1})$  and  $M_2 = (Q^{M_2}, A^{M_2}, \rightarrow_{M_2}, q_0^{M_2})$  be two IOLTSs such that  $A^{M_1} = A^{M_2}$ . The synchronous product of  $M_1$  and  $M_2$  is given by  $M_1 \parallel M_2 = (Q^{M_1 \parallel M_2}, A^{M_1 \parallel M_2}, \rightarrow_{M_1 \parallel M_2}, q_0^{M_1 \parallel M_2})$ , where  $Q^{M_1 \parallel M_2} = Q^{M_1} \times Q^{M_2}$ ,  $A^{M_1 \parallel M_2} = A^{M_1} = A^{M_2}$ ,  $q_0^{M_1 \parallel M_2} = (q_0^{M_1}, q_0^{M_2})$ , and  $\rightarrow_{M_1 \parallel M_2}$  is given by the following rules:

- $\forall a \in A^{M_1 \parallel M_2} : (p \xrightarrow{a}_{M_1} p') \wedge (q \xrightarrow{a}_{M_2} q') \vdash (p, q) \xrightarrow{a}_{M_1 \parallel M_2} (p', q')$
- $(p \xrightarrow{\tau}_{M_1} p') \vdash (p, q) \xrightarrow{\tau}_{M_1 \parallel M_2} (p', q)$
- $(q \xrightarrow{\tau}_{M_2} q') \vdash (p, q) \xrightarrow{\tau}_{M_1 \parallel M_2} (p, q')$

### III. SYNCHRONOUS TESTING

There are different ways of applying test cases to the IUT depending upon the test architecture. In synchronous testing, the tester and the IUT are located on the same computer, and they interact with each other by invoking each others' procedures. We now explain the principles of synchronous testing in the context of IOCO-test theory.

In practice, the specification is given in some specification language whose operational semantics can be explained in terms of IOLTS. So, we assume here that the specification is given to us as an IOLTS  $S = (Q^S, A^S, \rightarrow_S, q_0^S)$ . Contrary to the specification, the IUT is given to us as a black box, that is, we do not have any a priori information about the control structure of the IUT. Nevertheless, we assume that the IUT is a discrete system and therefore can be modeled by an IOLTS  $I = (Q^I, A^I, \rightarrow_I, q_0^I)$ . We make some structural assumptions about the given specification and IUT:

- 1) The given specification  $S$  is deterministic.
- 2) We know about the potential actions that the IUT can perform, that is, we assume that  $A_I^I = A_I^S$  and  $A_O^I = A_O^S$ .
- 3) There is no loop in  $S$  or  $I$  comprising only output symbols and internal actions. Formally, if  $M \in \{S, I\}$ , then  $\forall \sigma \in (A_O^M \cup \{\tau\})^+, \forall q \in Q^M : (q \xrightarrow{\sigma} q)_M$  is false. This restriction is quite justified, because its withdrawal would mean that the system can either hang indefinitely or exhibit an infinite spontaneous behavior. On the other hand, imposition of this restriction implies that there is at least one quiescent state in every loop of  $S$  and  $I$ .

We essentially want to probe whether the IUT is in conformance with the given specification. Conformance is formally defined by the relation *ioconf* over IOLTS, the set of all IOLTSs.

**Definition 2:**  $\forall I, S \in \text{IOLTS} : I \text{ ioconf } S \equiv \forall \sigma \in \text{Traces}(S) : \text{Out}(q_0^I \text{ after } \sigma) \subseteq \text{Out}(q_0^S \text{ after } \sigma)$

For a given specification, conformance with respect to the specification is a safety property of the IUT. Because, if the IUT  $I$  after a finite trace  $\sigma$  performs an output action  $x \in A_O^I$ , whereas the specification after  $\sigma$  does not, then the IUT is non conforming with respect to the specification.

Intuitively, the above definition suggests that a conforming IUT need not execute the same traces as its specification. An IUT can maintain conformance w.r.t to the specification by first executing a trace  $\sigma$  of the specification until a certain point, and then deviating by performing some input action which the specification does not perform, and thereafter performing absolutely anything. Based on this, we have the following result from [3], which reduces the problem of determining conformance to determining that each execution traces of the IUT is foreseen by the specification.

**Theorem 1:** For  $S, I \in \text{IOLTS} : I \text{ ioconf } S$  implies  $\text{Traces}(I) \subseteq \text{Traces}(S)$ , if and only if  $S$  is input complete.

Although there are many ways of making the given specification input complete, certainly not every method preserves the conformance/non-conformance relationship of the specification vis-a-vis the IUT. To this end, we have the following definition from [3], which can convert the given specification into an *equivalent* input-complete specification.

**Definition 3:** For any  $M \in \text{IOLTS}$ , we can define  $M^c = (Q^{M^c}, A^{M^c}, \rightarrow_{M^c}, q_0^{M^c})$  such that  $Q^{M^c} = Q^M \cup \{\hat{q}\}; A^{M^c} = A^M; q_0^{M^c} = q_0^M; \rightarrow_{M^c} = \rightarrow_M \cup \{(q, a, \hat{q}) | q \in Q^M \wedge a \in A_I^M \wedge \neg(q \xrightarrow{a}_M)\} \cup \{(\hat{q}, a, \hat{q}) | a \in A^M\}$ .

Informally, any IOLTS  $M$  can be converted into an input complete  $M^c$  by performing the following three steps in sequence: (1) Add a new state  $\hat{q}$ . (2) Add transitions from every state  $q$  to  $\hat{q}$  on every input symbol  $a$  in  $A_I^M$ , such that there is no other transition on  $a$  emanating from  $q$ . (3) Add self loops at the state  $\hat{q}$  on every symbol in  $A^M$ . Now, we have the following result from [3].

**Theorem 2:**  $\forall S, I \in \text{IOLTS} : I \text{ ioconf } S$  if and only if  $I \text{ ioconf } S^c$  if and only if  $\text{Traces}(I) \subseteq \text{Traces}(S^c)$ .

In other words, in order to check the conformance of the IUT  $I$  with respect to the given specification  $S$ , it is perfectly alright to convert  $S$  into  $S^c$ , and then try to determine whether each execution trace of  $I$  is also an execution trace of  $S^c$ .

Like the specification and the IUT, a test case is also modeled by an IOLTS. In addition, the IOLTS modeling a test case has some of its states marked with a verdict *fail*. Formally, a test case is a quintuple  $T = (Q^T, A^T, \rightarrow_T, q_0^T, V^T)$ , where  $V^T$  is a partial function from the set  $Q^T$  to the set  $\{\text{fail}\}$ . Each test case is characterized by three structural features: (1) A test case is always deterministic. (2) A test case is always finite, that is,  $|\text{Traces}(T)|$  is always a finite set. (3) The alphabet of a test case is always the mirror image of the alphabet of the IUT, that is,  $(A_I^T =$

$A_O^I \wedge (A_O^T = A_I^T)$ . This is because, the input (output) action of the IUT interacts with the output (input) action of the test case.

The synchronous execution of a test case  $T$  on the IUT  $I$  is modeled by the process  $I \parallel T$ . During the execution of a test case against the IUT, the tester knows the current action being executed by it as well as its current state. We are yet to explain how test cases are generated, howsoever we want every generated test case should be such that during the execution of a test case against the IUT if the tester finds itself in a state labeled *fail*, it should mean that the IUT is non conforming with respect to the specification, and therefore should be rejected. Formally, this can be stated as follows:

**Definition 4:**  $T$  rejects  $I$  if and only if  $\exists (p, q) \in Q^{I \parallel T} : q_0^{I \parallel T} \xrightarrow{*}_{I \parallel T} (p, q) \wedge (V^T(q) = \text{fail})$ .

A test suite is a collection of test cases. A desirable feature of a test suite is that it should be sound and exhaustive with respect to the specification. From sound we mean that a conforming IUT should not be rejected by any test case in the test suite, and from exhaustive we mean that a non conforming IUT should be rejected by at least one test case in the test suite. A test suite is complete, if it is both sound and exhaustive.

**Definition 5:** Given a test suite  $TS$  for a given specification  $S$ , (1)  $TS$  is sound if  $\forall I \in \text{IOLTS} : I \text{ iocnf } S \Rightarrow \forall T \in TS : \neg(T \text{ rejects } I)$ . (2)  $TS$  is exhaustive if  $\forall I \in \text{IOLTS} : \neg(I \text{ iocnf } S) \Rightarrow \exists T \in TS : T \text{ rejects } I$ . (3)  $TS$  is complete, if it is both sound and exhaustive.

In [5], the authors have proposed an algorithm to generate a complete test suite.

#### IV. ASYNCHRONOUS TESTING

Asynchronous testing is characterized by indirect interaction between the IUT and the tester through some medium. Formally, the medium is depicted by a pair of FIFO queues—the input queue and the output queue.

Now, let us explain the difference between synchronous testing and asynchronous testing with an example. Figure 1 shows a test case  $T$  and the IUT  $I$  interacting with each other asynchronously through a pair of queues.  $T$  and  $I$  interact with each other in a complementary fashion, which means that the input (output) queue of  $T$  is the output (input) queue of  $I$ . The IOLTS  $I$  depicts a coffee machine which (1) first accepts a coin through its input queue and then dispenses coffee through its output queue, or (2) dispenses tea through its output queue for free—strange though it sounds, but it happens when a company wants its newly launched product to reach out to masses. The IOLTS  $T$  depicts a typical user who after putting the coin into its output queue is ready to

accept either tea or coffee through its input queue. The state shown black in color is the only *fail* state of  $T$ . It is easy to see that  $T$  rejects  $I$  only when the interaction between them is asynchronous. Hence, the test suite developed for synchronous testing can not be used asynchronously. As will become clear in the next section, the reason for rejection of  $I$  in asynchronous testing is that the trace of events  $!tea.(?coin)^*$  when observed through a queue context could appear as  $?coin.!tea.(?coin)^*$ .

In [9] it was shown that asynchronous testing can be simulated by synchronous testing, provided the two queues are considered as an integral part of  $I$ . This was formalized by defining for each IOLTS  $M = (Q^M, A^M, \rightarrow_M, q_0^M)$  a corresponding IOLTS  $[M] = (Q^{[M]}, A^{[M]}, \rightarrow_{[M]}, q_0^{[M]})$ , where  $Q^{[M]} = (A_I^M)^* \times Q^M \times (A_O^M)^*$ ;  $A_I^{[M]} = A_I^M$ ,  $A_O^{[M]} = A_O^M$ ;  $q_0^{[M]} = (\epsilon, q_0^M, \epsilon)$ ; and  $\rightarrow_{[M]}$  is given by the following axioms and rules:

- (A<sub>1</sub>)  $\vdash \forall a \in A_I^M : (u, q, v) \xrightarrow{a}_{[M]} (ua, q, v)$
- (A<sub>2</sub>)  $\vdash \forall x \in A_O^M : (u, q, xv) \xrightarrow{x}_{[M]} (u, q, v)$
- (R<sub>1</sub>)  $(q \xrightarrow{\tau}_M q') \vdash (u, q, v) \xrightarrow{\tau}_{[M]} (u, q', v)$
- (R<sub>2</sub>)  $(q \xrightarrow{a}_M q') \vdash (au, q, v) \xrightarrow{\tau}_{[M]} (u, q', v)$
- (R<sub>3</sub>)  $(q \xrightarrow{x}_M q') \vdash (u, q, v) \xrightarrow{\tau}_{[M]} (u, q', vx)$

Intuitively, an IOLTS  $M$  interacting through a pair of queues with its environment appears as  $[M]$  to its environment. In any state  $(u, q, v)$  of  $[M]$ , the attributes  $u$  and  $v$  correspond to the contents of the input queue and output queue, respectively. Since both the input and output queues are presumably unbounded, it should be noted that  $[M]$  will always be an *infinite* state process even if  $M$  is a *finite* state process. It does not come as a surprise that there exists a correspondence between the traces executed by  $M$  and the traces executed by  $[M]$ . Formally, this correspondence can be described by the relation  $\lambda$  defined in terms of the prefix operator  $\preceq$  as follows:

**Definition 6:**  $\forall \sigma, \sigma' \in (A^M)^* : (\sigma, \sigma') \in \lambda$  iff  $(\sigma \downarrow_{A_O^M} = \sigma' \downarrow_{A_O^M}) \wedge (\sigma \downarrow_{A_I^M} \preceq \sigma' \downarrow_{A_I^M}) \wedge \forall w \preceq \sigma, \forall w' \preceq \sigma' : |w| = |w'| \Rightarrow w' \downarrow_{A^M} \preceq w \downarrow_{A^M}$ .

We have the following theorem from [7].

**Theorem 3:**  $\forall (\sigma, \sigma') \in \lambda : \sigma \in \text{Traces}(M)$  iff  $\sigma' \in \text{Traces}([M]')$ .

The above theorem states that a trace  $\sigma$  executed by an IOLTS  $M$  could appear as  $\sigma'$ , when  $M$  is observed through a queue context. Conversely, any trace observed as  $\sigma'$

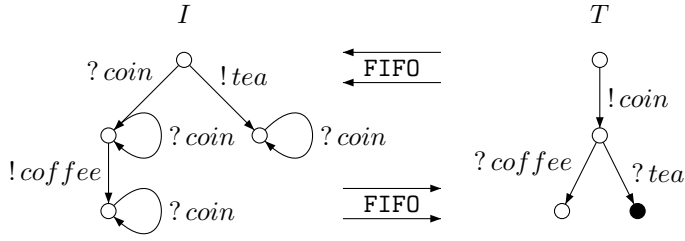


Figure 1. IUT  $I$  interacting asynchronously with a test case  $T$

implies that the actual trace executed by the system  $M$  is  $\sigma$ . It should be noted that the relation  $\lambda$  is a reflexive relation and therefore the actual trace  $\sigma$  could appear as  $\sigma$  also. We end this section by defining the notion of conformance for asynchronously communicating systems by the relation  $ioconf_A$  as follows:

**Definition 7:**  $\forall S, I \in \text{IOLTS} : I \text{ ioconf}_A S$  if and only if  $[I] \text{ ioconf } [S]$ .

We have the following Theorem from [3].

**Theorem 4:**  $\forall S, I \in \text{IOLTS} : I \text{ ioconf } S$  does not imply  $[I] \text{ ioconf } [S]$ , and at the same time  $[I] \text{ ioconf } [S]$  does not imply  $I \text{ ioconf } S$ .

To get around this negative result, we propose a tagging scheme (or protocol) in the next section.

## V. TAGGING

The tagging protocol is as simple as this. Each component of the asynchronously communicating distributed system before sending a message, say,  $x$  to one of its peer component, augments the message with a tag, whose value is either zero or one. Thus, instead of sending just  $x$ , it sends  $x(0)$  or  $x(1)$ . A component sends  $x(0)$  through its output queue when its input queue is *empty*, and sends  $x(1)$  when its input queue is *non-empty*.

The ability of a component to determine whether its input queue is empty is a fair assumption. In SPIN [1], a well known model checker, “Empty” is a predefined function that takes the name of the channel as an argument and returns “True”, if the number of messages currently held by the channel is zero, and returns “False” otherwise. This actually depicts a situation wherein each component of the message passing system has installed some message queuing system such as IBM’s Web Sphere MQ, Oracle Advanced Queuing, etc on it, and has a routine continuously running in the background that keeps ‘listening’ to the message queuing system.

In the context of our formal setup, if an asynchronously communicating distributed system implements the aforementioned tagging protocol, then the tagged asynchronous behavior of its component  $M = (Q^M, A^M, \rightarrow_M, q_0^M)$  can be given by  $[M]' = (Q^{[M]'}, A^{[M]'}, \rightarrow_{[M]'}, q_0^{[M]'})$ , where:

- $A_I^{[M]'} = \{a(t) \mid a \in A_I^M \wedge t \in \{0, 1\}\}$
- $A_O^{[M]'} = \{a(t) \mid a \in A_O^M \wedge t \in \{0, 1\}\}$
- $Q^{[M]'} = \{(u, q, v) \mid u \in (A_I^{[M]'})^*, q \in Q^M, v \in (A_O^{[M]'})^*\}$
- $q_0^{[M]'} = (\epsilon, q_0^M, \epsilon)$
- $\rightarrow_{[M]'}$  is given by the following axioms and rules:

$$(A'_1) \quad \vdash (u, q, \epsilon) \xrightarrow{a(0)}_{[M]'} (u.a(0), q, \epsilon)$$

$$(A'_2) \quad \vdash (u, q, v \neq \epsilon) \xrightarrow{a(1)}_{[M]'} (u.a(1), q, v)$$

$$(A'_3) \quad \vdash (u, q, x(0).v) \xrightarrow{x(0)}_{[M]'} (u, q, v)$$

$$(A'_4) \quad \vdash (u, q, x(1).v) \xrightarrow{x(1)}_{[M]'} (u, q, v)$$

$$(R'_1) \quad (q \xrightarrow{\tau}_M q') \vdash (u, q, v) \xrightarrow{\tau}_{[M]'} (u, q', v)$$

$$(R'_2) \quad (q \xrightarrow{a}_M q') \vdash (a(0).u, q, v) \xrightarrow{\tau}_{[M]'} (u, q', v)$$

$$(R'_3) \quad (q \xrightarrow{a}_M q') \vdash (a(1).u, q, v) \xrightarrow{\tau}_{[M]'} (u, q', v)$$

$$(R'_4) \quad (q \xrightarrow{x}_M q') \vdash (\epsilon, q, v) \xrightarrow{\tau}_{[M]'} (\epsilon, q', v.x(0))$$

$$(R'_5) \quad (q \xrightarrow{x}_M q') \vdash (u \neq \epsilon, q, v) \xrightarrow{\tau}_{[M]'} (u, q', v.x(1))$$

It should be noted that the axioms and rules to generate  $[M]$  and  $[M]'$  are absolutely same except that each axiom or rule for the former is split into two cases for the latter. The first case is when the input or output queue is empty and therefore the tag is “0”, and the second case is when the input or output queue is non-empty and therefore the tag is “1”.

**Lemma 1:**  $\forall a_1.a_2.\dots.a_n \in (A^M)^* : \text{If } (q_0^M \xrightarrow{a_1.a_2.\dots.a_n}_M q_1)$ , then  $(\epsilon, q_0^M, \epsilon) \xrightarrow{a_1(0).a_2(0).\dots.a_n(0)}_{[M]'} (\epsilon, q_1, \epsilon)$ .

*Proof* If any IOLTS  $M$  can make a transition  $q \xrightarrow{a}_M q'$  such that  $a \in A_I^M$ , then  $[M]'$  can make the corresponding transitions  $(\epsilon, q, \epsilon) \xrightarrow{a(0)}_{[M]'} (a(0), q, \epsilon) \xrightarrow{\tau} (\epsilon, q', \epsilon)$ . Similarly, if an IOLTS  $M$  can make a transition  $q \xrightarrow{x}_M q'$  such that  $x \in A_O^M$ , then  $[M]'$  can make the corresponding transitions  $(\epsilon, q, \epsilon) \xrightarrow{\tau}_{[M]'} (\epsilon, q', x(0)) \xrightarrow{x(0)} (\epsilon, q', \epsilon)$ . Thus by induction on  $n$  the left hand side implies the right hand side.

**Lemma 2:**  $\forall M_1, M_2 \in \text{IOLTS} : \text{If } \text{Traces}(M_1) \subseteq \text{Traces}(M_2)$ , then  $\text{Traces}([M_1]') \subseteq \text{Traces}([M_2]')$ .

## VI. TEST GENERATION

*Proof* The Axioms  $A'_1 - A'_4$  and the Rules  $R'_1 - R'_5$  constitute a mechanism (or an axiomatic system) to construct  $[M]'$  from the given  $M$ , step-by-step. A key feature of this mechanism is that the antecedent of each rule is a linear time property of the IOLTS  $M$ . Thus  $Traces(M_1) \subseteq Traces(M_2)$  implies that if a rule is applicable on  $M_1$ , then it is as well applicable on  $M_2$ . Hence, in the incremental construction of  $[M_1]'$  and  $[M_2]'$ , if a transition is added to  $[M_1]'$ , then it is as well added to  $[M_2]'$ . This proves the lemma.

We now state the main result of this paper as follows:

**Theorem 5:**  $\forall S, I \in IOLTS : I \text{ ioconf } S \text{ iff } Traces([I]') \subseteq Traces([S^c]')$ .

*Proof* Suppose, it is the case that  $I \text{ ioconf } S$ . By the Theorem 2 we have  $Traces(I) \subseteq Traces(S^c)$ . Finally, by the Lemma 2 we have  $Traces([I]') \subseteq Traces([S^c]')$ .

Now, let us prove the converse. Suppose, it is the case that  $\neg(I \text{ ioconf } S)$ . By the Theorem 2, it is also the case that  $\neg(I \text{ ioconf } S^c)$ . By the Definition 2,  $\exists a_1.a_2.\dots.a_n \in (A^S)^*, \exists x \in A_O^S : a_1.a_2.\dots.a_n.x \in Traces(I) \wedge a_1.a_2.\dots.a_n.x \notin Traces(S^c) \wedge a_1.a_2.\dots.a_n \in Traces(S^c)$ . By the Lemma 1, we have  $a_1(0).a_2(0).\dots.a_n(0).x(0) \in Traces([I]')$ .

Now, in what follows, we will try to prove that  $a_1(0).a_2(0).\dots.a_n(0).x(0) \notin Traces([S^c]')$ . Suppose that  $\sigma^t = a_1(t_1).a_2(t_2).\dots.a_n(t_n).x(t_x) \in Traces([S^c]')$  such that  $\forall i \in \{1, 2, \dots, n, x\} : t_i \in \{0, 1\}$ . By the Theorem 3,  $\exists \sigma.x \in Traces(S^c)$  such that  $(\sigma.x, a_1.a_2.\dots.a_n.x) \in \lambda$ . Knowing that  $\sigma.x \neq a_1.a_2.\dots.a_n.x$ , let us assume that  $w$  is the longest common prefix of  $\sigma.x$  and  $a_1.a_2.\dots.a_n.x$  such that  $w.x_k.\sigma'.x = \sigma.x$  and  $w.a_k.a_{k+1}.\dots.a_n.x = a_1.a_2.\dots.a_n.x$ . Now by virtue of Definition 6, it has to be the case that  $a_k$  is an input symbol, and  $x_k$  is an output symbol. Let  $q_0^{S^c} \xrightarrow{w}_{S^c} q_w \xrightarrow{x_k}_{S^c} q_{x_k} \xrightarrow{\sigma'}_{S^c} q_{\sigma'} \xrightarrow{x}_{S^c} q_x$ , then by the Lemma 1 we have  $(\epsilon, q_0^{S^c}, \epsilon) \xrightarrow{a_1(0).a_2(0).\dots.a_{k-1}(0)}_{[S^c]'}$   $(\epsilon, q_w, \epsilon) \xrightarrow{a_k(t_k).a_{k+1}(t_{k+1})\dots a_n(t_n).x(t_x)}_{[S^c]'}$   $(u, q_x, v)$ . We claim that since  $a_k \neq x_k$ , it cannot be the case that  $t_k, t_{k+1}, \dots, t_n, t_x$  are all equal to 0. Because,  $[S^c]'$  has two options at the state  $(\epsilon, q_w, \epsilon)$ . The first option is one in which  $[S^c]'$  first puts the  $x_k(0)$  into the output queue and latter performs the input action  $a_k(1)$ . Clearly, this option proves that  $a_1(0).a_2(0).\dots.a_n(0).x(0) \notin Traces([S^c]')$ . The second option is one in which  $[S^c]'$  first performs the input action  $a_k(0)$  and latter puts the output symbol  $x_k(1)$  into the output queue. In this case  $[S^c]'$  has to perform an output action  $x_k(1)$ , which again proves that  $a_1(0).a_2(0).\dots.a_n(0).x(0) \notin Traces([S^c]')$ .

Test generation comprises generating a set of test cases (test suite) from the given specification  $S$  such that the test cases when executed against  $[I]'$  should be able to conclude whether  $Traces([I]') \subseteq Traces([S^c]')$ . A test suite can be defined in a sequence of steps as follows:

- 1) Convert the given specification  $S$  into an equivalent input complete specification  $S^c$  according to the Definition 3.
- 2) Convert  $S^c$  into  $[S^c]' = (Q^{[S^c]'}, A^{[S^c]'}, \rightarrow_{[S^c]'}, q_0^{[S^c]'})$  using the Axioms  $A'_1 - A'_4$  and the Rules  $R'_1 - R'_5$ .
- 3) Convert  $[S^c]'$  into its deterministic equivalent  $([S^c]')_D = (Q^{([S^c]')_D}, A^{([S^c]')_D}, \rightarrow_{([S^c]')_D}, q_0^{([S^c]')_D})$ .
- 4) Convert  $([S^c]')_D$  into a canonical tester  $C = (Q^C, A^C, \rightarrow_C, q_0^C, V^C)$ , where  $Q^C = Q^{([S^c]')_D} \cup \{\check{q}\}; A_I^C = A_O^{([S^c]')_D}, A_O^C = A_I^{([S^c]')_D}; q_0^C = q_0^{([S^c]')_D}; \rightarrow_C = \rightarrow_{([S^c]')_D} \cup \{(q, a, \check{q}) | q \in Q^{([S^c]')_D} \wedge a \in A^{([S^c]')_D} \wedge \neg(q \xrightarrow{a})\};$  and  $V^C(\check{q}) = fail$ .

Informally,  $([S^c]')_D$  can be converted into  $C$  by (a) mirror imaging the input-output alphabet, (2) by adding a new state  $\check{q}$ , and (3) by adding transitions  $(q, a, \check{q})$  from state  $q$ , if there is no other transition on the symbol  $a$  from the state  $q$ .

- 5) It should be noted that the canonical tester obtained in the above step fails if and only if  $\neg(Traces([I]') \subseteq Traces([S^c]'))$ . Hence,  $C$  is a complete test suite in itself. However owing to its large size, it is practically infeasible to execute all the finite behaviors of  $C$ . Therefore, as a final step, we generate a test suite  $TS$  comprising all the finite behaviors of the  $C$ .

It should be noted that the above stated points only define the required test suite, however it is not possible to generate the test suite following those steps. Because,  $[S^c]', ([S^c]')_D$  and  $C$  are all infinite state processes even if the given specification  $S$  is a finite state process. Now, we describe a methodology to generate the canonical tester  $C$  incrementally, that is, state-by-state from the given specification  $S^c = (Q^{S^c}, A^{S^c}, \rightarrow_{S^c}, q_0^{S^c})$ . The generation of the test cases is all the same except that generation is stoped after certain time to ensure that the test cases are finite.

The generation of  $C$  makes use of the Algorithm 1, given in Figure 2, which computes the set  $(X \text{ after } \epsilon)_{[S^c]'}$   $X \subseteq Q^{[S^c]'}$ .

**Algorithm 1.**

```

while True do
   $Y := X$ 
  if  $((\epsilon, q, v) \in X) \wedge (q \xrightarrow{S^c} q') \wedge (x \in A_O^{S^c})$  then
     $X := X \cup (\epsilon, q', v.x(0))$  /* This step is due to rule  $R'_4$  */
  end if
  if  $((u \neq \epsilon, q, v) \in X) \wedge (q \xrightarrow{S^c} q') \wedge (x \in A_O^{S^c})$  then
     $X := X \cup (u, q', v.x(1))$  /* This step is due to rule  $R'_5$  */
  end if
  if  $((a(k).u, q, v) \in X) \wedge (q \xrightarrow{S^c} q') \wedge (a \in A_I^{S^c}) \wedge k \in \{0, 1\}$  then
     $X := X \cup (u, q', v)$  /* This step is due to rules  $R'_2, R'_3$  */
  end if
  if  $((u, q, v) \in X) \wedge (q \xrightarrow{S^c} q')$  then
     $X := X \cup (u, q', v)$  /* This step is due to rule  $R'_1$  */
  end if
  if  $(Y = X)$  then
    return( $Y$ )
  end if
end while

```

Figure 2. An algorithm to compute  $(X \text{ after } \epsilon)_{[S^c]}$  s.t  $X \subseteq Q^{[S^c]}$ **Initial state of  $C$  :**

$$q_0^C = ((\epsilon, q_0^{S^c}, \epsilon) \text{ after } \epsilon)_{[S^c]}.$$

**Output transitions of  $C$  :**

- 1)  $\forall X \in Q^C : X \xrightarrow{a(0)}_C X' = \{(u.a(0), q, \epsilon) | (u, q, \epsilon) \in X\} \text{ after } \epsilon)_{[S^c]}.$
- 2)  $\forall X \in Q^C : X \xrightarrow{a(1)}_C X' = \{(u.a(1), q, v \neq \epsilon) | (u, q, v) \in X\} \text{ after } \epsilon)_{[S^c]}.$

**Input transitions of  $C$  :**

- 1)  $\forall X \in Q^C : X \xrightarrow{x(0)}_C X' = \{(u, q, v) | (u, q, x(0).v) \in X\} \text{ after } \epsilon)_{[S^c]}.$
- 2)  $\forall X \in Q^C : X \xrightarrow{x(1)}_C X' = \{(u, q, v) | (u, q, x(1).v) \in X\} \text{ after } \epsilon)_{[S^c]}.$

**Fail transitions of  $C$  :**

- 1)  $\forall X \in Q^C : X \xrightarrow{a(0)}_C \checkmark$  iff  $\exists(u, q, \epsilon) \in X.$

- 2)  $\forall X \in Q^C : X \xrightarrow{a(1)}_C \checkmark$  iff  $\exists(u, q, v \neq \epsilon) \in X.$
- 3)  $\forall X \in Q^C : X \xrightarrow{x(0)}_C \checkmark$  iff  $\exists(u, q, x(0).v) \in X.$
- 4)  $\forall X \in Q^C : X \xrightarrow{x(1)}_C \checkmark$  iff  $\exists(u, q, x(1).v) \in X.$

## VII. CONCLUSION

Testing is the commonest way of verifying the correctness of a system with respect to its specification. In model based testing, the system under test is described by some model, which serves as a basis for test case generation. IOLTS is one such model, which is broadly used to describe the functional behavior of an interactive system. Through this paper, we have addressed the problem of generating test cases for testing a component of an asynchronously communicating distributed system, which is modeled by an IOLTS. Unlike synchronous testing, generating a complete test suite for asynchronous testing is always difficult, because the asynchronous behavior is observed through an unbounded queue context and is therefore always infinite. Generally, asynchronous conformance bears no relation with the synchronous conformance. What we have proved in this paper is that if the distributed system implements the tagging protocol, synchronous conformance starts bearing a relation with the asynchronous conformance. Based on this relation, we showed how to generate a complete test suite with respect to the given specification.

## REFERENCES

- [1] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
- [2] C. Jard and T. Jéron. TGV: theory, principles and algorithms, 2002.
- [3] Claude Jard, Thierry Jéron, Lénaïck Tanguy, and César Viho. Remote testing can be as powerful as local testing. In *FORTE*, pages 25–40, 1999.
- [4] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
- [5] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
- [6] J. Tretmans and E. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, Nuremberg, Germany, pages 31–43, 2003.
- [7] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.

- [8] Machiel van der Bijl and Fabien Peureux. I/O-automata based testing. In *Model-Based Testing of Reactive Systems*, pages 173–200, 2004.
- [9] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In *Protocol Test Systems*, pages 55–66, 1992.
- [10] M. Weiglhofer and F. Wotawa. Asynchronous input-output conformance testing. In Sheikh Iqbal Ahamed, Elisa Bertino, Carl K. Chang, Vladimir Getov, Lin Liu, Hua Ming, and Rajesh Subramanyan, editors, *COMPSAC (1)*, pages 154–159. IEEE Computer Society, 2009.