Distributed Testing of Multi Input/Output Transition System [*]

Zhongjie Li, Xia Yin and Jianping Wu
Dep. of Computer Science and Technology, Tsinghua University
Rm.401, East Main Bldg., Beijing, P.R.China, 100084
lzj@csnet1.cs.tsinghua.edu.cn

Abstract

This paper develops the refusal testing theory of multi input/output transition system (MIOTS) in the direction of distributed testing where multiple testers are involved. Centralized MIOTS testing (where only one tester is used) can be based on two types of observers: the singular-observer and the all-observer. For each of the two cases, we define a test architecture and propose a method to distribute a centralized test case onto a set of distributed testers. The singular-observer can only observe one channel at a time, and the distribution of singular-observer tests is indeed a projection of the global test tree on local testers with proper handover messages. The all-observer can observe all the output channels simultaneously, and distributing all-observer tests requires a mechanism for solving output contentions and synchronizing local testers. Examples are used to illustrate these methods.

1. Introduction

Conformance testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the IUT (implementation under test), and based on observations made during the execution of the tests, a verdict about the correct functioning of the implementation is given. In formal conformance testing it is assumed that we have a formal specification, and implementations whose behavior is also formally modeled but not apriori known. Labeled Transition Systems (LTS) is a well-known model [7] for describing processes such as specifications, implementations and tests. LTS does not distinguish input action and output action, which is not very realistic. Multi input/output transition system (MIOTS) [3] is an extension model to LTS, which clearly distinguishes inputs and outputs, explicitly models interface distribution, and allows of input blocking. MIOTS refusal testing unifies existing LTS-based testing theories in a single framework. However, it assumes that experiment on implementations can access all interfaces of the implementation, which is not always the case in reality. Especially when the implementation itself is distributed over geographically dispersed places, an observer may not be able to observe all interfaces of the implementation. Instead, the observer is distributed over many "smaller" observers that each has access to a limited number of interfaces of the implementation, this is well-known as distributed testing [1]. Comparatively, testing that engages only one observer to record and carry out all the testing activities dispersed on several interfaces is classified as centralized testing [1]. In this paper, we use "observer" as a synonym of "tester", which can issue stimulations as well as absorb responses, not just "observe".

There are two ways to get distributed tests: generating from the specification directly, or translating from centralized tests that have been generated from the specification. This paper follows the second way, giving solid algorithms that automatically distribute centralized tests into distributed tests. Then we just need to pay attention to designing centralized tests, which are translated into distributed tests automatically at the time when a distributed test architecture is required. There have been plenty of researches on the distributed testing with no coordination procedures between testers (e.g. [6]). These works cannot avoid the well-known control-observation problem, which decreases the testing power, so in this paper we confine ourselves to the study of distributed testing with coordination between testers, viz. coordinated distributed testing. It is aimed to have equal testing power as the centralized testing.

Multi-port FSM with n ports (np-FSM) is one of the most heavily used models for distributed testing [1]. However, it is a special case of the MIOTS model, and methods for np-FSM cannot be applied to MIOTS. For one thing, np-FSM is based on the asynchronous communication paradigm whereas MIOTS assumes synchronous com-

[*] This work was supported by the 973 project No.2003CB314801 and NSFC project No.90104002, No.60102009

munication. Besides, a transition of np-FSM is labeled by a pair of actions: $\langle input, outputs\rangle$ whereas that of MIOTS is labeled by only one action. These differences require different treatments in the distribution algorithm: (1) The distribution of np-FSM tests should ensure that inputs are sent to the implementation in the right order and outputs are causally related to the stimulating input; the distribution of MIOTS tests must synchronize the testing activities action by action and ensure that every action is in the right order. (2) In a transition of np-FSM, the order of the outputs at different ports is not cared (the outputs occur independently and concurrently); in MIOTS, the order of the output at different ports is relevant and should be tested. Therefore, different distribution algorithms are needed for MIOTS.

[2] mentioned the situation where distributed MIOTS testing is necessary, but no proposal has been presented since then. So our work may be the first. For queue systems (which models a system embedded in a test context of FIFO queues, and can be related to MIOTS models in the way proposed in [9]), [4] proposed an automatic distribution method to test the conformance relation **ioconf**. We notice that the centralized tests for queue systems are very similar to the *all-observer* tests for MIOTS [5]. We make significant modifications to the method in [4], adapting it to the distributed all-observer testing for MIOTS. The whole approach develops the refusal testing theory of MIOTS in the direction of distributed testing.

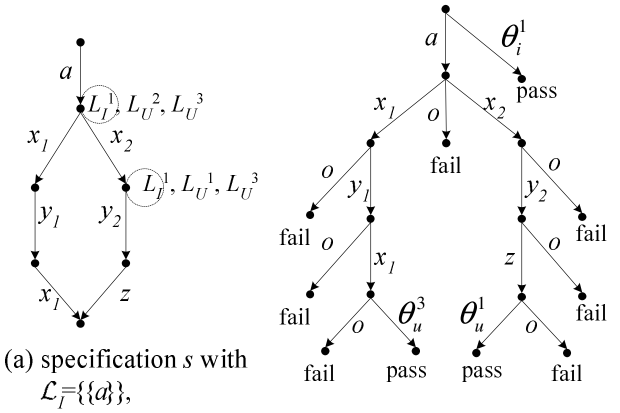
This paper is organized as follows. Section 2 fixes notations and recapitulates the refusal testing for MIOTS. Section 3 studies the distribution of centralized singular-observer tests, and Section 4 discusses the distribution of centralized all-observer tests. Concluding remarks and future works are presented in Section 5.

2. Refusal testing for MIOTS

2.1. Multi input/output transition system

Definition 1 *A (labeled) transition system over* L *is a quadruple* $\langle S, L, \rightarrow, s_0\rangle$ *where* S *is a (countable) set of states,* L *is a (countable) set of observable actions,* $\rightarrow \subseteq S \times L \times S$ *is a set of transitions, and* $s_0 \in S$ *is the initial state.*

We denote the class of all transition systems over L by $\mathcal{LTS}(L)$. The observable behavior of a transition system is expressed using sequences consisting of actions and sets of refused actions, i.e., sequences in $(L \cup P(L))^*$ ($P(L)$ is the power-set of L). Such sequences are called *failure traces*, comparable to *traces* (those in L^*). *A refusal transition* is defined as a self-loop transition in the form $s \xrightarrow{A} s'$ where $A \subseteq L$ is called *a refusal* of s, meaning that the system is unable to perform any action in A from state s.



(a) specification s with $\mathcal{L}_I=\{\{a\}\}$, $\mathcal{L}_U=\{\{x_1,x_2\},\{y_1,y_2\},\{z\}\}$ (b) a centralized $\mathcal{SOBS}$ test t

Figure 1. an MIOTS and its $\mathcal{SOBS}$ test

Definition 2 *Let* $p \in \mathcal{LTS}(L)$, *then*

1. $init(p) =_{def} \{\alpha \in L | \exists p' : p \xrightarrow{\alpha} p'\}$
2. $der(p) =_{def} \{p' | \exists \sigma \in (L \cup P(L))^* : p \xrightarrow{\sigma} p'\}$
3. $ftraces(p) =_{def} \{\sigma \in (L \cup P(L))^* : p \xrightarrow{\sigma}\}$
4. $pref(\sigma_2) = \{\sigma_1 | \exists \sigma' : \sigma_1 \cdot \sigma' = \sigma_2$ *and* $\sigma_1, \sigma_2, \sigma' \in (L \cup \mathcal{P}(L))^*\}$
5. P **after** $\sigma =_{def} \{p' | \exists p \in P : p \xrightarrow{\sigma} p'\}$
6. p *is deterministic iff* $\forall \sigma \in L^* : |\{p\}$ **after** $\sigma| \leq 1$
7. p *is output-finite if there is a natural number* N *s.t.* $\forall p' \in der(p)$, *the set* $X = \{\sigma_u \in (L_U)^* | p' \xrightarrow{\sigma_u}\}$ *is finite and* $\forall \sigma_u \in X : |\sigma_u| \leq N$, *where* L_U *is the set of output actions.*

Definition 3 *A multi input/output transition system* p *over partitioning* $\mathcal{L}_I = \{L_I^1, \ldots, L_I^n\}$ *of* L_I *and partitioning* $\mathcal{L}_U = \{L_U^1, \ldots, L_U^m\}$ *of* L_U *is a transition system with inputs and outputs,* $p \in \mathcal{LTS}(L_I \cup L_U)$, *such that for all* $L_I^j \in \mathcal{L}_I, \forall p' \in der(p)$, *if* $\exists a \in L_I^j : p' \xrightarrow{a}$ *then* $\forall b \in L_I^j : p' \xrightarrow{b}$. *The universe of multi input/output transition systems over* $\mathcal{L}_I$ *and* $\mathcal{L}_U$ *is denoted by* $\mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U)$.

EXAMPLE 1 Figure 1a shows a specification $s \in \mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U)$. It has one input channel and three output channels. Initially, s can accept the input a; then it produces either x_1, y_1 and x_1 sequentially, or x_2, y_2 and z sequentially. After that s becomes stable and no more output can be produced. Self-loop transitions are refusal transitions. (Note that not all the refusal transitions are shown.)

2.2. Singular-observer and implementation relation

Implementation relation is defined between implementations and specifications to answer the basic testing problem: what do we mean by saying that an implementation conforms to (or, implements) the specification? Refusal testing [8] is one implementation relation where experiments are not only able to detect whether actions can occur, but also able to detect whether actions can fail, i.e. refused by the system. Refusal testing theory has been developed for MIOTS in [3]. In MIOTS refusal testing, $\theta_i^j (j = 1, \ldots, n)$ are added to observe the inability of the implementation to accept an input action at the channel L_I^j (input suspension, denoted by a special label ξ^j), and $\theta_u^k (k = 1, \ldots, m)$ are added to observe the inability to produce outputs at the channel L_U^k (output suspension, denoted by a special label δ^k). Let $\Theta = \{\theta_i^1, \ldots, \theta_i^n, \theta_u^1, \ldots, \theta_u^m\}$ denote all the suspension detection labels. Now, implementations that are modeled as members of $\mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U)$ are observed by observers modeled in $\mathcal{MIOTS}(\mathcal{L}_U^\theta, \mathcal{L}_I^\theta)$ where $\mathcal{L}_I^\theta = \{L_I^1 \cup \{\theta_i^1\}, \ldots, L_I^n \cup \{\theta_i^n\}\}$, $\mathcal{L}_U^\theta = \{L_U^1 \cup \{\theta_u^1\}, \ldots, L_U^m \cup \{\theta_u^m\}\}$.

Singular observers are a special class of MIOTS observers. They consist of finite, serial compositions of providing a single input action at some channel L_I^j and detection of its acceptance or rejection, and observing some channel L_U^k and detection of the occurrence or absence of outputs produced at this channel. The set of all singular observers over $\mathcal{L}_I$ and $\mathcal{L}_U$ is denoted by $\mathcal{SOBS}(\mathcal{L}_U^\theta, \mathcal{L}_I^\theta)$.

In correspondence with the observations defined on $(L_I \cup L_U \cup \Theta)^*$, we define the suspension traces of p to be its failure traces restricted to $(L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$: $straces(p) =_{def} ftraces(p) \cap (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$. Responses of the implementation after a specific suspension trace that can be observed by singular observers (outputs, input and output suspensions) are collected into the set out. For s in Example 1, we have $out(s \textbf{ after } a) = \{x_1, x_2, \xi^1, \delta^2, \delta^3\}$.

Definition 4 *The relation* $\textbf{mioco}_\mathcal{F} \subseteq \mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U) \times \mathcal{LTS}(L_I, L_U)$, *where* $\mathcal{F} \subseteq (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$, *is defined by*
$i \textbf{ mioco}_\mathcal{F}\ s =_{def} \forall \sigma \in \mathcal{F} : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$.

[2] gives an algorithm to derive a set of tests from a specification such that these tests are able to reject implementations that are $\textbf{mioco}_\mathcal{F}$-incorrect, and accept implementations that are $\textbf{mioco}_\mathcal{F}$-correct. A test is exactly a singular observer with ending states associated with either the verdict **pass** or the verdict **fail**, which can be used to give an indication about the (in)correctness of implementations when the test is running against implementations.

Definition 5 *A test t is a singular observer $t \in \mathcal{SOBS}(\mathcal{L}_U^\theta, \mathcal{L}_I^\theta)$ with two special verdict states* **pass** *and* **fail***:* $init(t') = \emptyset$ *iff* $t' = \textbf{pass}$ *or* $\textbf{fail}$.

For the specification s in Figure 1a with respect to the relation $\textbf{mioco}_\mathcal{F}$, the $\mathcal{SOBS}$ test t checking $(a - L_U^1, a \cdot x_1 - L_U^2, a \cdot x_2 - L_U^2, a \cdot x_1 \cdot y_1 - L_U^1, a \cdot x_2 \cdot y_2 - L_U^3, a \cdot x_1 \cdot y_1 \cdot x_1 - L_U^3, a \cdot x_2 \cdot y_2 \cdot z - L_U^1)$ is shown in Figure 1b. $\sigma - L_U^k$ represents checking the output behavior of the IUT on channel L_U^k after having performed the trace σ. The label o matches all the other actions on the same channel, excluding those explicitly designated. The test runs as follows: first it inputs a to the implementation i; if a is accepted, we observe the channel L_U^1 , and await an output. Then, if x_1 or x_2 is produced, the test continues; otherwise (another output, or no output is produced) the test ends with the **fail** verdict. This way ensures that if $out(i \textbf{ after } a) \not\subseteq out(s \textbf{ after } a)$, the error will be detected.

2.3. All-observer

The $\mathcal{SOBS}$ test suite tends to be very large because the singular observer can only observe one output channel at a time in the testing process, and thus separate tests are needed to check each output channel after each $\sigma \in \mathcal{F}$. To reduce the size of the test suites, we have developed a theory of all-observer testing [5]. The all-observer is also a special class of observers. They can observe all the output channels simultaneously.

Definition 6 *An all-observer u over $\mathcal{L}_I$ and $\mathcal{L}_U$ is a finite, deterministic MIOTS $u \in \mathcal{MIOTS}(\mathcal{L}_U^\theta, \mathcal{L}_I^\theta)$ such that*

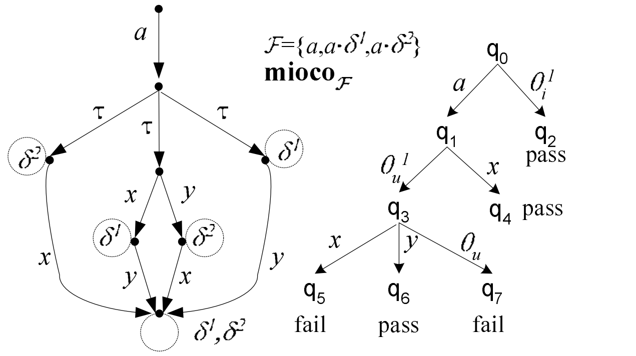
$\forall u' \in der(u) : init(u') = \emptyset$ *or* $init(u') = L_U \cup \{\theta_u\}$
or $init(u') = \{a, \theta_i^j\}$
for some $j \in \{1, \ldots, n\}$ *and* $a \in L_I^j$
or $init(u') = L_U^k \cup \{\theta_u^k\}$ *for some* $k \in \{1, \ldots, m\}$

the set of all-observer over $\mathcal{L}_I$ and $\mathcal{L}_U$ is denoted by $\mathcal{AOBS}(\mathcal{L}_U^\theta, \mathcal{L}_I^\theta)$.

The special label θ_u is used to detect the situation that none of the output channels of the implementation can produce output. Then, an $\mathcal{AOBS}$ test is exactly an all-observer with ending states associated with either the verdict **pass** or **fail**.

Definition 7 *An all-observer test is defined as* $t = \langle Q, L_I \cup L_U \cup \Theta \cup \{\theta_u\}, \rightarrow, q_0 \rangle \in \mathcal{AOBS}(\mathcal{L}_U^\theta, \mathcal{L}_I^\theta)$ *such that for each* $q \in Q : init(q) = \emptyset$ *iff* $q = \textbf{pass}$ *or* $q = \textbf{fail}$.

Thus one $\mathcal{AOBS}$ test may replace many $\mathcal{SOBS}$ tests by simultaneously checking all the output channels after each $\sigma \in \mathcal{F}$. We have proved in [5] that for a special kind of implementations (modeled by strace-reorderable and output-finite MIOTS) and a trace set $\mathcal{F}$ satisfying a given condition, the $\mathcal{SOBS}$ test suite can be replaced by an $\mathcal{AOBS}$



(a) an $\mathcal{MIOTS}(\mathcal{L}_I,\mathcal{L}_U)$ q (b) a centralized $\mathcal{AOBS}$ test t

Figure 2. an MIOTS and its $\mathcal{AOBS}$ test

test suite of smaller size without compromising the testing power.

EXAMPLE 2. Figure 2a shows a specification $q \in \mathcal{MIOTS}(\{\{a\}\}, \{\{x\}, \{y\}\})$, where a is an input action and x, y are output actions. δ^1, δ^2 denotes the refusal transitions on L_U^1 and L_U^2, respectively. s can perform the input action a, and then exhibit three possible output behaviors non-deterministically. τ denotes the internal action, which expresses the nondeterminism of the output behavior. The $\mathcal{AOBS}$ tests t checking the output behavior of an implementation after $a \cdot \delta^1$ is shown in Figure 2b.

3. Distribution of $\mathcal{SOBS}$ test cases

3.1. Distributed test architecture

Testing of a system with distributed interfaces can be either centralized or distributed. In centralized testing (see Figure 3a), only one tester is used and the tester has a full control over all the channels of the IUT (implementation under test). The correspondent test cases are called centralized test cases (or simply centralized tests). In distributed testing (see Figure 3b), several testers are used and they interact with the IUT in parallel. Each tester can access only a subset of the channels. The correspondent test cases are called distributed test cases (or simple distributed tests). For sake of clarity, we consider only the case that each tester is associated with each channel. The extension to a general channel mapping is straightforward.

As Figure 3b shows, each tester connects with one channel (labeled as p_1, p_2, p_3, $\ldots$) of the IUT for input or output. Then for an MIOTS with n input and m output channels, the number of distributed testers is $m + n$. The tester R_j provides inputs at p_j (the actions belong to L_I^j, $j = 1, \ldots, n$), and R_{n+k} provides inputs at p_{n+k} (the actions belong to L_U^k, $k = 1, \ldots, m$). Testers coordinate their testing activities through asynchronous coordination chan-

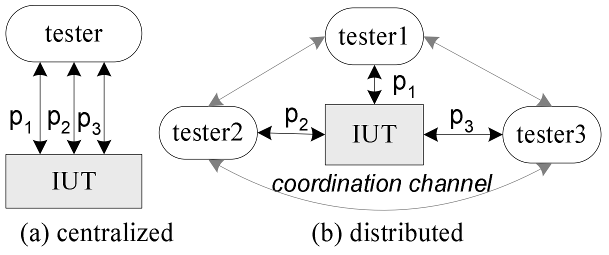


Figure 3. centralized vs distributed test architecture

nels. Let $Sync\_Set$ be the collection of coordination messages, $R_i!sync$ denotes the sending of a coordination message $sync \in Sync\_Set$ to R_i, and $R_i?sync$ denotes the reception of a coordination message $sync \in Sync\_Set$ from R_i. All the actions of sending and receiving of coordination messages constitute the set $Sync\_Actions$. The behavior of each tester $R_i (i = 1, \ldots, m + n)$ is denoted by $t(R_i)$, and a distributed test is represented by $t_c = \langle t(R_1), \ldots, t(R_{m+n})\rangle$.

Definition 8 *$t(R_k)$ is an LTS: $\langle T(R_k), L(R_k), \rightarrow_{R_k}, t_0(R_k)\rangle$ where:*

∘ $t(R_k)$ is finite and deterministic.

∘ $T(R_k)$ is a finite set of states, including **pass** *and* **fail** *for test verdicts: $init(\mathbf{pass}) = init(\mathbf{fail}) = \emptyset$.*

∘ $L(R_k)$ is the set of actions defined on
$$\begin{cases} L_I^k \cup \{\theta_i^k\} \cup Sync\_Actions & if\ k \le n \\ L_U^{k-n} \cup \{\theta_u^{k-n}\} \cup Sync\_Actions & if\ k > n \end{cases}$$

∘ $\forall t' \in T(R_k)$ and $t' \notin \{\mathbf{pass}, \mathbf{fail}\}$, either $\exists a \in L_I^k : init(t') = \{a, \theta_i^k\}$, or $init(t') = L_U^{k-n} \cup \{\theta_u^{k-n}\}$, or $init(t') \subseteq Sync\_Actions$.

∘ $t_0(R_k)$ is the initial state.

Obviously $t(R_k)$ has a tree-type transition graph, rooted at $t_0(R_k)$. We define below $p\_next(t)$ to be the next active channel on any state t of the centralized test t. The tester in charge of $p\_next(t)$ is called the active tester for state t, the others are inactive testers.

$$p\_next(t) = \begin{cases} p_j & if\ init(t) = \{a, \theta_i^j\} \text{ and } a \in L_I^j \\ p_{k+n} & if\ init(t) = L_U^k \cup \{\theta_u^k\} \\ null & if\ init(t) = \emptyset \end{cases}$$

3.2. Distribution algorithm

Next, we give a recursive algorithm that converts a centralized test t to its distributed correspondence $t_c = \langle t(R_1), \ldots, t(R_{m+n})\rangle$.

Algorithm 1 $visit(t, t_1, \ldots, t_{m+n})$
input: an $\mathcal{SOBS}$ centralized test t
output: the $\mathcal{SOBS}$ distributed test t_c

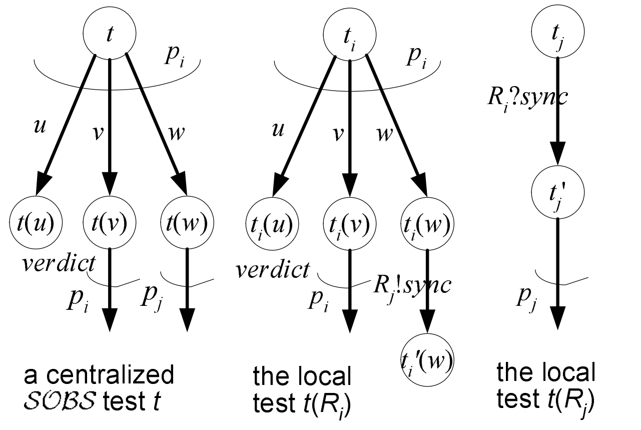


Figure 4. Illustration of the $\mathcal{SOBS}$ distribution algorithm

```
initialization: $sync := 0$, $t(R_i) := \mathbf{stop}$ for $i = 1, \ldots, m+n$
parameters: $t$ denotes the current state of the test $t$, and $t_i$ denotes the current state of $t(R_i)$
(* the next channel to be observed: $p_i$ *)
$p_i := p\_next(t)$
1. extend $t(R_i)$: replicate all the outgoing transitions of state $t$ at state $t_i$, $t(v)$ denotes the end state of the $v$ transition in test $t$, $t_i(v)$ is a newly added state in test $t(R_i)$
    $\forall v : t \xrightarrow{v} t(v), \text{let } t_i \xrightarrow{v} t_i(v)$
2. for each action $v \in init(t)$ such that $t \xrightarrow{v} t(v)$
2.1 if $p\_next(t(v)) = null$, then $t(v)$ is a verdict state. Assign this verdict to the current state of $t(R_i) - t_i(v)$:
    $t_i(v) := t(v)$
2.2. if $p_j = p\_next(t(v))$ and $j = i$, call the recursive procedure using the new current states:
    $visit(t(v), \ldots, t_i(v), \ldots, t_{m+n})$
2.3. if $p_j = p\_next(t(v))$ and $j \neq i$, add coordination messages sync to state $t_i(v)$ and state $t_j$, and then call the recursive procedure using the new current states:
    $t_i(v) \xrightarrow{R_j!sync} t_i'(v); t_j \xrightarrow{R_i?sync} t_j'$;
    $inc(sync)$;
    $visit(t(v), \ldots, t_i'(v), \ldots, t_j', \ldots, t_{m+n})$
```

Figure 4 diagrams the running of this algorithm. Each call of the procedure $visit(t, t_1, \ldots, t_{m+n})$ is used to process a state and the associated outgoing transitions of the centralized test t. Each time a transition is processed, the behavior of the relevant testers are expanded. Suppose the next active channel is p_i, then the transitions should be handled by tester R_i, as step (1) shows. Next, each outgoing transition in state t is scanned and three cases are distinguished. If the current state $t(v)$ is a verdict state, step (2.1) terminates the recursive procedure. The same verdict is assigned to tester R_i. Step (2.2) handles the case when two consecutive active channels are the same and thus no coordination message is needed. Then the procedure is called recursively with the new current states. If the active channel p_j following that transition is different with p_i, a coordination message should be added between tester R_i and R_j, as step (2.3) shows. Coordination messages are identified by different integers. We call $visit(t_0, t_0(R_1), \ldots, t_0(R_{m+n}))$ ($t_0(R_i)$ denotes the initial state of $t(R_i)$) to get the distributed test t_c. It can be seen from this algorithm that although all the testers run in parallel, their interactions with the IUT are sequential: there is only one active tester at any time. This guarantees the distributed test has the same test behavior as the centralized test. Verdicts can be emitted in any tester during the testing; in this case the other testers should be terminated, perhaps by means of timer operations or auxiliary coordination procedures.

EXAMPLE 3. Figure 5 shows the distributed test $t_c = \langle t(R_1), t(R_2), t(R_3), t(R_4) \rangle$ derived by Algorithm 1 for the centralized $\mathcal{SOBS}$ test t in Figure 1b. Coordination messages are distinguished by their identifiers k as $sync(k)$. The tester R_1 is in charge of the input channel L_I^1 ; R_2, R_3 and R_4 observe responses at the output channel L_U^1, L_U^2 and L_U^3, respectively. After R_1 has sent the input a, it notifies the next active tester - R_2 by the message $R_2!sync(0)$. R_2 will then await outputs at L_U^1. Depending on the output, it either sends the message $sync(1)$ or $sync(4)$ to R_3, or arrives at a **fail** verdict. Such a procedure continues until a verdict is made, and then the test case is stopped.

4. Distribution of $\mathcal{AOBS}$ test cases

In this section, we discuss the problem of distributed $\mathcal{AOBS}$ testing. First, we introduce some notations. Then we consider the following issues in turn: the distributed test architecture, the distribution rule, the election service (which plays a key role in synchronizing multiple testers, like a global clock), and the differences of our work from [4]. Finally, we give an example of the distribution.

4.1. Notations

Some additional notations are needed:

1. The name of the channel to which the action u belongs is denoted by $port(u)$.

$$port(u) =_{def} \begin{cases} p_j & if\ u \in L_I^j \cup \{\theta_i^j\} \\ p_{n+k} & if\ u \in L_U^k \cup \{\theta_u^k\} \end{cases}$$

2. The set of channels that are observed in a state q of a centralized $\mathcal{AOBS}$ test is denoted by $p\_init(q)$: $p\_init(q) = \{p_k | \exists u \in init(q) : port(u) = p_k\}$.

3. If $k > n$, $ready(k)$ is a predicate that becomes true if the tester R_k has made sure that p_k can produce an output in L_U^{k-n} ; and $ready(k, \theta_u^{k-n})$ is another predicate that

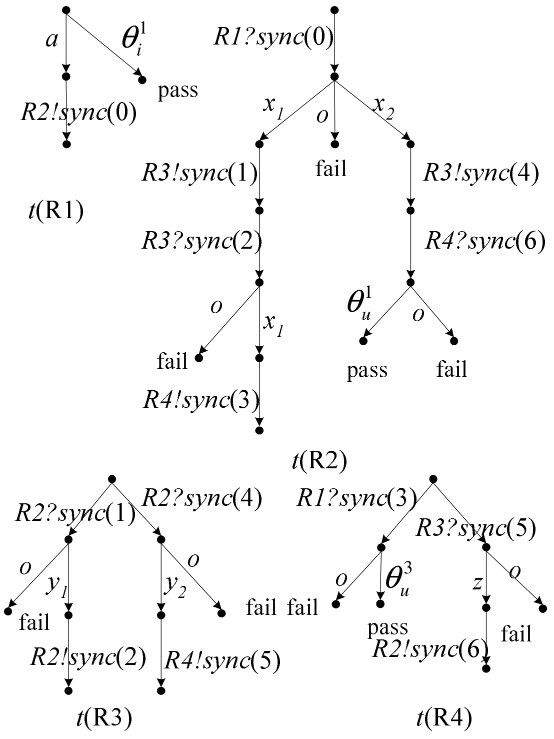


Figure 5. the distributed $\mathcal{SOBS}$ test t_c

becomes true if $ready(k)$ is not true for a reasonable time-period. If $k < n$, $ready(k)$ is always true.

A centralized $\mathcal{AOBS}$ test case has two types of states except the verdict states: single-channel states and all-output-channel states, viz. $\forall q \in Q$: either $p\_init(q) = \{p_i\}$, $i \in \{1, \ldots, m+n\}$ (meaning that the test provides an input at channel L_I^j or awaits an output at channel L_U^k in state q), or $p\_init(q) = \{p_{n+1}, \ldots, p_{n+m}\}$ (meaning that the test observes all the output channels in state q). Testers in $p\_init(q)$ are called *active testers*, the others are *inactive testers*.

4.2. Distributed test architecture

Distribution of an $\mathcal{AOBS}$ test case on several testers has special requirements and cannot share the same mechanism with the $\mathcal{SOBS}$ tests. Reference [4] proposed an approach that uses the test architecture in Figure 6. Compared with that of distributed $\mathcal{SOBS}$ tests, this test architecture adds a component called *election service* that communicates synchronously with all the distributed testers via a synchronous channel (named CS). In $\mathcal{AOBS}$ testing, the tester in an all-output-channel state does not block any output channel of the IUT, i.e., it is prepared to accept any output that is ready to occur at any of the output channels under observation. Therefore, an arbitration method is needed to select one channel from all the ready channels to receive an out-

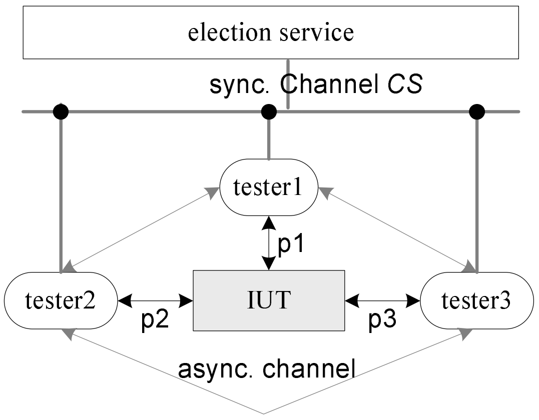


Figure 6. the distributed test architecture in [4]

put; otherwise, distributed testers may do outputs in a disordered and uncontrollable manner. In [4], each tester keeps watch on its channel, and when observing that the channel is ready to produce an output, it applies to the election service. The election service, possibly from several applications, elects one and notifies the election result to all the testers. Then the elected tester performs the ready output; all the other testers update their views of the current global testing state using the election result. In this way, all the testers can run in a synchronized manner. Step by step, each tester knows clearly how the testing is going on.

The definition of election service and distribution rules should guarantee that the interactions made by the distributed test case with the IUT are the same as those made by the centralized test case with the IUT, i.e. $traces(t_c \| i) = traces(t_c \| i)$. This ensures they have equal testing power in accepting correct implementations and rejecting incorrect implementations. The election service is only needed at the all-output-channel states, because in the single-channel states only one channel is observed and no arbitration is needed. The election strategy can be round-robin or whatever is fair to all the testers. We will describe the abstract model of the election service later.

4.3. Distribution rule

Below we give the rule used to distribute a centralized $\mathcal{AOBS}$ test t onto $m+n$ distributed testers, resulting in a distributed test t_c. It consists of three parts, which are quite different with those in [4] as explained later. Starting from the initial state of t - q_0, for any state q and its outgoing transitions, the distribution rule synthesizes a set of transitions for each distributed tester R_i. The part A and B deals with the all-output-channel states. Part-A defines the transitions of the active testers $\{R_i | p_i \in p\_init(q)\}$ (see Fig-

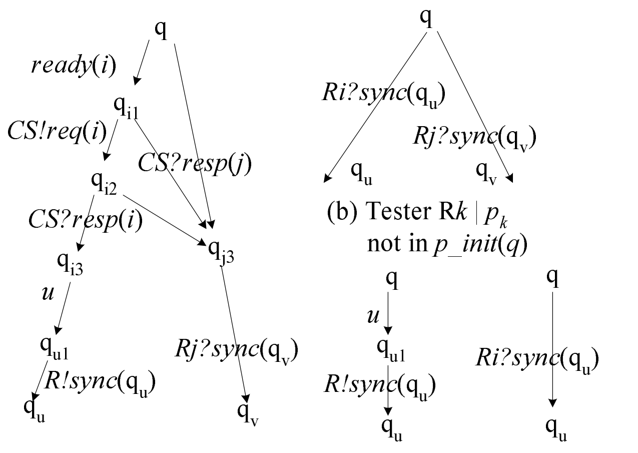


Figure 7. the distribution rule illustration

ure 7). Part-B defines the transitions of the inactive testers $\{R_k | p_k \notin p\_init(q)\}$ (see Fig.7b). Part-C deals with single-channel states: $p\_init(q) = \{p_i\}$, and defines the transitions of R_i (see Fig.7c) and other testers (see Fig.7d).

[A] $\dfrac{\forall p_i \in p\_init(q):}{q \xrightarrow{ready(i)} q_{i1} \xrightarrow{CS!req(i)} q_{i2} \xrightarrow{CS?req(i)} q_{i3}}$

$\forall u \in init(q) | port(u) = p_i, q \xrightarrow{u} q_u$:

- $q_{i3} \xrightarrow{u} q_{u1} \xrightarrow{R!sync(q_u)} q_u$

$\forall p_j \in p\_init(q)$ and $j \neq i$:

- $q \xrightarrow{CS?resp(j)} q_{j3}, q_{i1} \xrightarrow{CS?resp(j)} q_{j3}, q_{i2} \xrightarrow{CS?resp(j)} q_{j3}$
- $\forall v \in init(q) | port(v) = p_j, q \xrightarrow{v} q_v$: $q_{j3} \xrightarrow{R_j?sync(q_v)} q_v$

[B] $\dfrac{\forall p_k \notin p\_init(q)}{\forall p_i \in p\_init(q) : q \xrightarrow{R_i?sync(q_u)} q_u}$

[C] $p\_init(q) = \{p_i\}$,

$\forall u \in init(q) | port(u) = p_i, q \xrightarrow{u} q_u$:

$q \xrightarrow{u} q_{u1} \xrightarrow{R!sync(q_u)} q_u$

$\forall p_k \notin p\_init(q) : q \xrightarrow{R_i?sync(q_u)} q_u$

where R is the set of all the testers except R_i

Explanation

[A] When observing that the channel p_i is ready to produce an output $(ready(i))$, R_i sends an application $CS!req(i)$ to the election service and then awaits the election result. If it is elected as indicated by receiving $CS?resp(i)$, R_i performs the output, and then according to the actual output action u, informs all the other testers of the current global testing state q_u by sending coordination messages $R!sync(q_u)$. If another tester R_j is elected as indicated by receiving $CS?resp(j)$, R_i should update its view of the global testing state using the notification $R_j?sync(q_v)$ from R_j.

[B] In an all-output-channel state q, inactive testers - $\{R_k | p_k \notin p\_init(q)\}$ should await the coordination message and update their view of the global testing state accordingly.

[C] In a single-channel state q, there is no need to use the election service because only one tester is active. The tester performs input or output immediately and informs all the other testers of the current global state by $R!sync(q_u)$.

Such a distribution rule ensures that the distributed test has the same test behavior as the centralized test. In fact, their externally-observable parts are trace-equivalent (c.f. [4] for an approximate proof).

Special treatment: In the all-output-channel state, there is a special action θ_u whose occurrence cannot be observed by one tester solely but can only be observed by the active testers altogether. Before applying the distribution rule, we should preprocess the centralized test case as follows: every θ_u transition is replaced by m transitions labeled by the actions θ_u^k $(k = 1, \ldots, m)$ respectively. These m transitions all end at the same verdict state q_θ to which θ_u has led. Special treatment is also needed for the election service. When R_{n+k} observes that could occur, it applys to the election service by sending a special application $CS!req(n+k, \theta_u^k)$. On receiving such an application, the election service saves it and starts a special election process. If a normal application $CS!req(j)$ arrives, it will override all the saved θ_u^k applications and j is elected. If all θ_u^k $(k = 1, \ldots, m)$ applications have arrived, the election service will elect one of the active testers and broadcast the election result: $CS?resp(n + l)$ for some $l \in \{1, \ldots, m\}$.

4.4. Election service

Figure 8 shows the abstract model of the election service. It accepts applications from several testers and broadcasts the election result to all the active testers. The transition labeled by $|||_{i \in P_i} CS?req(i)$ specifies all possible interleaving of the requests coming from the testers $\{R_i | i \in P_i\}$ and is actually a hypercube (where $P_1, \ldots, P_k$ are all possible subsets of the set $[n + 1, n + m]$); the transition labeled by $|||_{i \in [n+1,n+m]} CS!resp(f(P_i))$ is another hypercube, which specifies the broadcasting of the election result to all the active testers (f is the election function). The transition labeled by $|||_{j \in [1,m]} CS?req(n + j, \theta_u^j)$ specifies all possible interleaving of the special requests coming from the testers $\{R_i | i \in [n + 1, n + m]\}$; the transition labeled by $|||_{j \in [n+1,n+m]} CS!resp(n + l)$ specifies the broadcasting of the election result to all the active testers. The transition labeled by $|||_{j \in P} CS?req(n+j, \theta_u^j) \cup CS?req(i)$ specifies all possible interleaving of the special requests coming from the testers $\{R_j | j \in P \subseteq [n + 1, n + m]\}$ plus

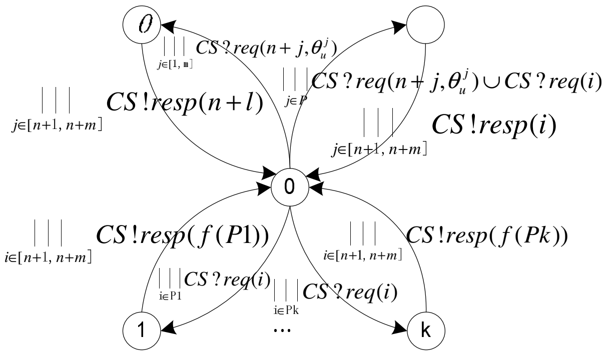

$\emptyset$
$|||_{j\in[1,m]} CS?req(n+j,\theta_u^j)$
$|||_{j\in P} CS?req(n+j,\theta_u^j) \cup CS?req(i)$
$|||_{j\in[n+1,n+m]} CS!resp(n+l)$
$|||_{j\in[n+1,n+m]} CS!resp(i)$
0
$|||_{i\in[n+1,n+m]} CS!resp(f(P1))$
$|||_{i\in[n+1,n+m]} CS!resp(f(Pk))$
$|||_{i\in P1} CS?req(i)$
$|||_{i\in Pk} CS?req(i)$
1
…
k

Figure 8. the election service

a normal request from a tester R_i; the transition labeled by $|||_{j\in[n+1,n+m]}CS!resp(i)$ specifies the broadcasting of the election result to all the active testers.

The implementation of the election service in asynchronous communication is much more complex. Suppose testers send applications, and the election service notifies the result both through messages, then the problems of incomplete-election and wrong-election may arise. Incomplete-election means that an election is made from some of the applications while other applications are queued. This problem can be avoided by let the election service start a time waiting for some period before election (however, this time is hard to choose). If the queued applications enter the next election round and mix with new applications, wrong-election will occur. This problem can be avoided by numbering the applications and election results; obsolete applications are simply discarded by the election service, and an election result is only used by local testers if it matches the application just sent. In short, the implementation of the election service in asynchronous communication is much more complex than in synchronous communication. Therefore we have assumed the synchronous method.

4.5. Comparison with [4]

Work for distributed $\mathcal{AOBS}$ testing in this paper differs from that in [4] in the following aspects.

1. Both the specification model and the conformance relation are different. [4] is limited to the distributed testing of queue systems for the relation **ioconf**, while this paper studies the distributed testing of MIOTS for the relation $\mathbf{mioco}_{\mathcal{F}}$. Because queue systems and **ioconf** are special cases of MIOTS and $\mathbf{mioco}_{\mathcal{F}}$, respectively, our work is more general.

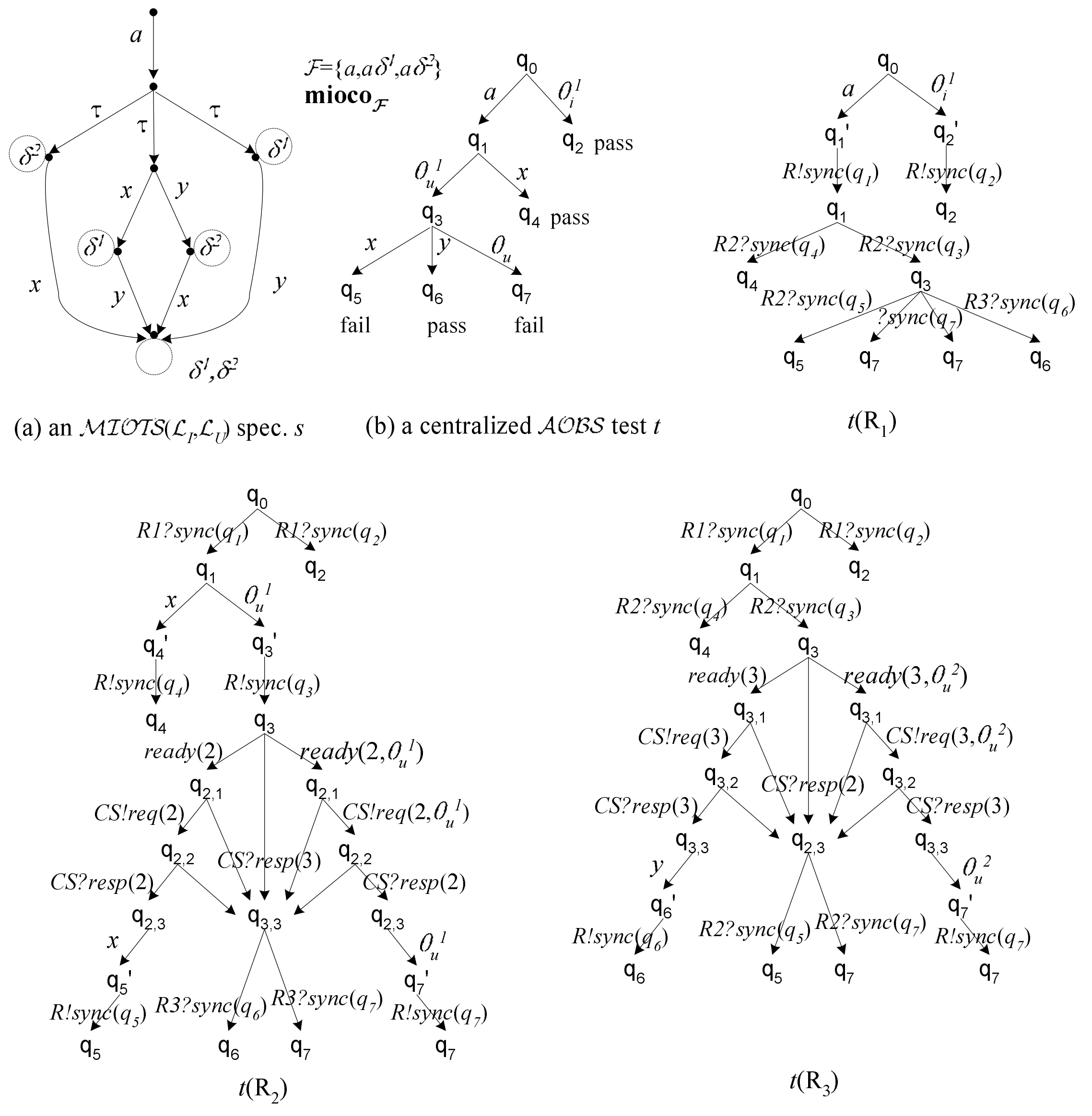
2. The definitions of the ready predicate are different. [4] defines a similar $present(a)$, which is evaluated true if a is an input action, or a is an output action ready in channel p_k ($a \in L_U^{k-n}$). Such a definition assumes that a tester can observe which output would be produced before actually receiving it. This assumption holds in queue systems but not in all the MIOTS systems. In many cases, a tester cannot observe which output would be produced before making the receiving action; it only knows that some output action can occur (we can assume that each output channel has a light that is turned on when the channel is ready to produce an output). The definition of $ready(i)$ in this paper is appropriate for these general cases and also applicable to the queue systems. Accordingly, the distribution rule and the election service are different from those in [4] in the usage of the ready/present predicate and the application messages.

3. The distribution rule in this paper distinguishes two types of global states: single-channel and all-output-channel. Single-channel states do not use the election service and thus follow a simpler distribution rule. [4] does not make this discrimination.

4. [4] treats θ_u as a normal output action in evaluating the $present(a)$ predicate. This is wrong as has been analyzed in Section 4.3. θ_u needs special treatments both before applying the distribution rule and in using the election service.

4.6. An example of distribution

EXAMPLE 4. Figure 9 shows the distributed test converted from the centralized $\mathcal{AOBS}$ test t in Figure 9b. As the specification q has three channels, the distributed test is performed by three testers R_1, R_2 and R_3. R_1 provides input and the other two await output. Suppose in a test run, the path ($q_0 - q_1 - q_3 - q_7$, or $a \cdot \theta_u^1 \cdot \theta_u$, see Figure 9b) is executed. The testing process runs as follows. First, R_1 sends the input a. Then, after observing that a is accepted, R_1 sends a state synchronization message $sync(q_1)$ to all the other testers, and becomes quiet. On receiving $sync(q_1)$, R_2 and R_3 update their view of the global testing state to q_1. Here, only channel L_U^1 is observed, by R_2. Observing that the implementation produces no output, R_2 executes θ_u^1, transfers to state q_3', and then notifies the other testers by sending $sync(q_3)$. Now, the test enters an all-output-channel observing mode, R_2 and R_3 both become active in observing channel L_U^1 and L_U^2 , respectively. As the implementation will not produce any output, $ready(2, \theta_u^1)$ and $ready(3, \theta_u^2)$ will both become true. Then R_2 and/or R_3 apply to the election service which, say, elects R_2. After this election result is disseminated, and also a series of local transitions and global synchronizations happen, R_1,R_2 and R_3 will all reach state q_7. The test ends with a **fail** verdict.



(a) an $\mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U)$ spec. s (b) a centralized $\mathcal{AOBS}$ test t $t(R_1)$

(c) the distributed $\mathcal{AOBS}$ test $t_c=(t(R_1), t(R_2), t(R_3))$

Figure 9. a specification and its $\mathcal{AOBS}$ test and the distribution

5. Conclusion and future work

Distributed testing of distributed systems has become an important area of conformance testing. There have been a lot of work on the distributed testing of the multi-port finite state machine with n ports (np-FSM) model. We haven't seen, however, any work on the distributed testing of the more general model: multi input/output transition system (MIOTS). So this paper aims to develop the refusal testing theory of MIOTS in the direction of distributed testing.

Centralized MIOTS testing (using only one tester) can be based on either the singular-observer or the all-observer. For each of the two cases, we define a test architecture and propose a method to distribute a centralized test case onto a set of distributed testers. Singular observers access one channel after another of the IUT and never observe more than one channel simultaneously. To distribute a singular-observer test, we just activate the testers one by one to communicate with the IUT. The testing process is really like a relay race: the turn is relayed from one tester to the other according to the order predefined in the centralized test. Essentially this distribution method amounts to a projection of the centralized test on several testers with appropriate handover messages. In contrast, the all-observer can observe all the output channels at one time, so an arbitration method is needed to select one channel from all the ready channels to do an output. Therefore, the distribution of all-observer tests requires a complex coordination approach. The arbitration is accomplished by a component called election service, which supervises the behavior of all the active testers, resolves output competition and ensures that the output action is performed in an ordered and controllable manner.

The distributed tests are usually very lengthy and complicate, so converting a centralized test case to its distributed correspondence manually tends to be tedious and error-prone. Therefore, automatic distribution methods can be very useful in practice. We are working now to optimize the proposed $\mathcal{AOBS}$ distribution rule to reduce the coordination overheads. One possible optimization lies in the fact that the global testing state need not be notified to all the local testers; only the next active testers need this information, while those inactive testers can be left blind until they will become active subsequently. We are also working on the test generation algorithms that derive distributed tests from the specification directly; such algorithms will save much time if we only test the implementation in a distributed test architecture.

References

[1] L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *Information and Software Technology*, 41(11-12):767–780, September 1999.

[2] L. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[3] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. *Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII'97)*, pages 23–38, 1997.

[4] C. Jard, T. Jeron, and H. Kahlouche. Towards automatic distribution of testers for distributed conformance testing. *FORTE/PSTV'98*, November 1998.

[5] Z. Li, J. Wu, and X. Yin. Testing multi input/output transition system with all-observer. *Testcom'2004*, pages 95–111, March 2004.

[6] G. Luo, G. Bochmann, R. Dessouli, P. Venkataram, and A. Ghedamsi. Test generation for the distributed test architecture. *IEEE Singapore Intl. Conf. on Networks/Intl. Conf. on Info. Eng.'93 (SICON/ICIE'93)*, pages 6–11, 1993.

[7] R. D. Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, (34):83–133, 1984.

[8] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

[9] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. *PSTV'92*, pages 131–145, 1992.