Implementation relations for the distributed test architecture*

Robert M. Hierons[1], Mercedes G. Merayo[1], and Manuel Núñez[2]

[1] Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex, UB8 3PH United Kingdom, rob.hierons@brunel.ac.uk,
mgmerayo@fdi.ucm.es
[2] Universidad Complutense de Madrid, Madrid, Spain, mn@sip.ucm.es

Abstract. Some systems interact with their environment at a number of physically distributed interfaces called ports. When testing such a system under test (SUT) it is normal to place a local tester at each port and the local testers form a local test case. If the local testers cannot interact with one another and there is no global clock then we are testing in the distributed test architecture. In this paper we explore the effect of the distributed test architecture when testing an SUT against an input output transition system, adapting the **ioco** implementation relation to this situation. In addition, we define what it means for a local test case to be deterministic, showing that we cannot always implement a deterministic global test case as a deterministic local test case. Finally, we show how a global test case can be mapped to a local test case.

1 Introduction

If the system under test (SUT) has physically distributed interfaces, called ports, then in testing we place a tester at each port. If we are applying black-box testing, these testers cannot communicate with each other, and there is no global clock then we are testing in the distributed test architecture [1]. It is known that the use of the distributed test architecture reduces test effectiveness when testing from a deterministic finite state machine (DFSM) and this topic has received much attention (see, for example, [2–6]). It is sometimes possible to use an external network, through which the testers can communicate, to overcome the problems introduced when testing from a DFSM. However, there are costs associated with deploying such a network and it is not always possible to run test cases that have timing constraints [7].

Previous work on testing in the distributed test architecture has focussed on testing from DFSMs, two effects being identified. First, *controllability* problems

* This research was carried out while the first author was visiting Universidad Complutense de Madrid under a grant *Programa de visitantes distinguidos e investigadores extranjeros en la UCM (Grupo Santander)*. Research partially supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01) and the Marie Curie project MRTN-CT-2003-505121/TAROT.

may occur, where a tester cannot know when to apply an input. Let us suppose, for example, that a test case starts with input x_p at port p, this should lead to output y_p at p only and this is to be followed by input x_q at $q \neq p$. The tester at q cannot know when x_p has been applied, since it does not observe either the input or output from this transition. The second issue is that there may be *observability* problems that can lead to fault masking. Let us suppose, for example, that a test case starts with input x_p at p, this is expected to lead to output y_p at p only, this is to be followed by input x_p at p and this should lead to output y_p at p and y_q at $q \neq p$. The tester at p expects to observe $x_p y_p x_p y_p$ and the tester at q expects to observe y_q. This is still the case if the SUT produces y_p and y_q in response to the first input and y_p in response to the second input: Two faults have masked one another in the sequences used but could lead to failures in different sequences. Work on testing in the distributed test architecture has largely concerned finding test sequences without controllability or observability problems (see, for example, [2–4, 6]) but recent work has characterized the effect of the distributed test architecture on the ability of testing to distinguish between a DFSM specification and a DFSM implementation [8].

While DFSMs are appropriate for modelling or specifying several important classes of system, they are less expressive than input output transition systems (IOTSs). For example, an IOTS can be nondeterministic and this is potentially important since distributed systems are often nondeterministic. In addition, in a DFSM input and output alternate and this need not be the case in IOTS. This paper investigates the area of testing from an IOTS in the distributed test architecture. For the sake of clarity, we focus on the case where there are two ports U and L, but our framework can be easily extended to cope with more ports. The paper explores the concept of a local test case (t_U, t_L), in which t_U and t_L are local testers at ports U and L respectively, and how we can assign verdicts. We adapt the well known **ioco** implementation relation [9, 10] by defining a new implementation relation **dioco** and prove that i **dioco** s for SUT i and specification s if and only if i can fail a certain test run when testing in the distributed test architecture. While **ioco** has been adapted in a number of ways (see, for example, [11–17]), as far as we know this is the first paper to define an implementation relation based on **ioco** for testing from an IOTS in the distributed test architecture. However, an implementation relation **mioco** has been defined for testing from an IOTS with multiple ports when there is a single tester that controls and observes all of the ports [18].

Interestingly, **ioco** and **dioco** are incomparable if we do not require the specification to be input enabled but otherwise we have that **dioco** is weaker that **ioco**. We define what it means for a local test case (t_U, t_L) to be deterministic and show that there are deterministic global test cases that cannot be implemented using deterministic local test cases. We also show how a global test case t can be mapped to a local test case that implements t, where this is possible. In effect, the notion of a local test case being deterministic captures what it means for a test case to be controllable while **dioco** describes the ability to distinguish between processes and so captures observability problems.

This paper is structured as follows. In Section 2 we give preliminary material and in Section 3 we define local test cases and what it means to pass or fail a test run. Section 4 gives the new implementation relation **dioco** and Section 5 defines what it means for a local test case to be deterministic and shows how given a test case t we can find a local test case that implements t. In Section 6 conclusions are drawn.

2 Preliminaries

In this section we present the main concepts used in the paper. First, we define input output transition systems and notation to deal with sequences of actions that can be performed by a system. After that we will comment on the main differences, with respect to *classical* testing, when testing in the distributed architecture.

2.1 Input output transition systems

An input output transition system is a labelled transition system in which we distinguish between input and output. We use this formalism to define processes.

Definition 1. *An input output transition system s, in short IOTS, is defined by (Q, I, O, T, q_{in}) in which Q is a countable set of states, $q_{in} \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$, where τ represents internal (unobservable) actions, is the transition relation. A transition (q, a, q') means that from state q it is possible to move to state q' with action $a \in I \cup O \cup \{\tau\}$. We let $\mathcal{IOTS}(I, O)$ denote the set of IOTSs with input set I and output set O.*

We say that the state $q \in Q$ is quiescent *if from q it is not possible to produce output without first receiving input. We say that the process s is* input enabled *if for all $q \in Q$ and $?i \in I$ there is some $q' \in Q$ such that $(q, ?i, q') \in T$. We say that the process s is* output-divergent *if it can reach a state in which there is an infinite loop that contains outputs and internal actions only.*

Given action a and process s, $a.s$ denotes the process that performs a and then becomes s. Given a countable set S of processes, $\sum S$ denotes the process that can nondeterministically choose to be any one of the processes in S. Given processes $s_1, \ldots, s_k$, we have that $s_1 || s_2 || \ldots || s_k$ is the process in which $s_1, \ldots s_k$ are composed in parallel and interact by synchronizing on common labels.

In this paper we use I for input and O for output rather than I and U, which are traditionally used in the work on IOTS. This is because U denotes the upper tester in protocol conformance testing and so, in this paper, U and L are used to denote ports. In order to distinguish between input and output we usually precede the name of a label with ? if it is an input and ! if it is an output. We assume that implementations are input enabled. This is a usual condition to ensure that implementations will accept any input provided by the tester. In

addition we also assume that all the processes considered in this paper, either implementations or models, are not output-divergent.

It is normal to assume that it is possible for the tester to determine when the SUT is quiescent and this is represented by δ. We can extend the transition relation to include in quiescent states *transitions* labelled by δ.

Definition 2. *Let (Q, I, O, T, q_{in}) be an IOTS. We can extend T, the transition relation, to T_δ by adding the transition (q, δ, q) for each quiescent state q. We let $\mathcal{A}ct$ denote the set of observable actions, that is, $\mathcal{A}ct = I \cup O \cup \{\delta\}$. A* trace *is an element of $\mathcal{A}ct^*$. Given a trace σ we let $in(\sigma)$ denote the sequence of inputs from σ. This can be recursively defined by the following in which ϵ is the empty sequence: $in(\epsilon) = \epsilon$, if $z \in I$ then $in(z\bar{z}) = zin(\bar{z})$ and if $z \notin I$ then $in(z\bar{z}) = in(\bar{z})$.*

Traces are often called *suspension traces*, since they can include quiescence, but since these are the only types of traces we consider we simply call them traces. Let us remark that traces are in $\mathcal{A}ct^*$ and so cannot contain τ. The following is standard notation in the context of **ioco** (see, for example, [9]).

Definition 3. *Let $s = (Q, I, O, T, q_{in})$ be an IOTS. We use the following notation.*

1. *If $(q, a, q') \in T_\delta$, for $a \in \mathcal{A}ct \cup \{\tau\}$, then we write $q \xrightarrow{a} q'$.*
2. *We write $q \stackrel{a}{\Longrightarrow} q'$, for $a \in \mathcal{A}ct$, if there exist $q_0, \ldots, q_m$ and $k \geq 0$ such that $q = q_0$, $q' = q_m$, $q_0 \xrightarrow{\tau} q_1, \ldots q_{k-1} \xrightarrow{\tau} q_k$, $q_k \xrightarrow{a} q_{k+1}$, $q_{k+1} \xrightarrow{\tau} q_{k+2}, \ldots, q_{m-1} \xrightarrow{\tau} q_m$.*
3. *We write $q \stackrel{\epsilon}{\Longrightarrow} q'$ if there exist $q_1, \ldots, q_k$, for $k \geq 1$, such that $q = q_1$, $q' = q_k$, $q_1 \xrightarrow{\tau} q_2, \ldots q_{k-1} \xrightarrow{\tau} q_k$.*
4. *We write $q \stackrel{\sigma}{\Longrightarrow} q'$ for $\sigma = a_1 \ldots a_m \in \mathcal{A}ct^*$ if there exist $q_0, \ldots, q_m$, $q = q_0$, $q' = q_m$ such that for all $1 \leq i < m$ we have that $q_i \stackrel{a_{i+1}}{\Longrightarrow} q_{i+1}$.*
5. *We write $s \stackrel{\sigma}{\Longrightarrow}$ if there exists q' such that $q_{in} \stackrel{\sigma}{\Longrightarrow} q'$ and we say that σ is a* trace *of s.*

Let $q \in Q$ and $\sigma \in \mathcal{A}ct^*$ be a trace. We consider

1. *q **after** $\sigma = \{r \in Q | q \stackrel{\sigma}{\Longrightarrow} r\}$*
2. *$out(q) = \{!O \in O | q \stackrel{!O}{\Longrightarrow}\}$*

The last function can be extended to deal with sets in the expected way: Given $Q' \subseteq Q$ we define $out(Q') = \cup_{q \in Q'} out(q)$.

We say that s is deterministic *if for every trace $\sigma \in \mathcal{A}ct^*$ we have that $out(q_{in}$ **after** $\sigma)$ contains at most one element.*

Let us remark that for any state q we have $q \Rightarrow q$ and that $q \xrightarrow{a} q'$ implies $q \stackrel{a}{\Longrightarrow} q'$. Let us also note that for all process s the empty sequence ϵ is a trace of s. While we do not require the models or implementations to be deterministic it is normal to use test cases that are deterministic. Next we present the standard implementation relation for testing from an IOTS [9, 10].

Definition 4. *Given processes i and s we have that i **ioco** s if for every trace σ of s we have that $out(i$ **after** $\sigma) \subseteq out(s$ **after** $\sigma)$.*

2.2 Multi-port input output transition systems

There are two standard test architectures [1], shown in Figure 1. In the local test architecture a global tester interacts with all of the ports of the SUT. In the distributed test architecture there are ports through which a system interacts with its environment. We focus on the case where there are two ports, traditionally called U and L for the ports connected to the upper and lower testers, respectively. In this paper we usually use the term IOTS for the case where there are multiple ports and when there is only one port we use the term single-port IOTS.

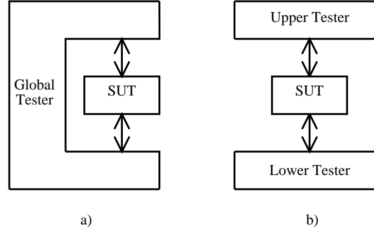

Fig. 1. The local and distributed test architectures

For an IOTS (Q, I, O, T, q_{in}) with ports U and L we partition the set I into sets I_U and I_L of inputs that can be received at U and L respectively and we define a set O_U of output that can be received at U and a set O_L of output that can be received at L. Each element of O is thus in $(O_U \cup \{-\}) \times (O_L \cup \{-\})$ in which $-$ denotes empty output and $(-, -)$ is not an element of O. We assume that the sets I_U, I_L, O_U, O_L are pairwise disjoint. Given $y = (!O_U, !O_L) \in O$ we let $y|_U$ denote $!O_U$ and $y|_L$ denote $!O_L$. If an output tuple has output $!O_p$ at one port p only then we often simply represent this as $!O_p$.

In order to apply tests, if we have a *global tester* that observes all of the ports then it observes a trace in $\mathcal{A}ct^*$, called a *global trace*. Given port p and a global trace $\bar{z}$ we let $\pi_p(\bar{z})$ denote the projection of $\bar{z}$ onto p and this is called a *local trace*. This is defined by the following rules in which $q \neq p$:

1. $\pi_p(\epsilon) = \epsilon$
2. if $z \in (I_p \cup O_p \cup \{\delta\})$ then $\pi_p(z\bar{z}) = z\pi_p(\bar{z})$
3. if $z = (!O_U, !O_L)$, $!O_p \neq -$, then $\pi_p(z\bar{z}) = !O_p\pi_p(\bar{z})$
4. if $z \in (I_q \cup O_q)$ or $z = (!O_U, !O_L)$ for $!O_p = -$ then $\pi_p(z\bar{z}) = \pi_p(\bar{z})$.

Given traces σ and σ' we write $\sigma \sim \sigma'$ if σ and σ' cannot be distinguished when making local observations, that is, $\pi_U(\sigma) = \pi_U(\sigma')$ and $\pi_L(\sigma) = \pi_L(\sigma')$. When testing in the distributed test architecture, each local tester observes a sequence of actions at its port. We cannot distinguish between two traces if they have the same projections at each port. That is why we cannot compare traces by equality but by considering the permutations of traces defined by using $\sim$.

3 Test cases for the distributed test architecture

Before discussing implementation relations for the distributed test architecture it is necessary to adapt the standard definition of a test case. In this paper a test case t is an IOTS with the same input and output sets as s. We also have that s and t synchronize on values and we say that such a test case is a *global test case*. Thus, if s is in $\mathcal{IOTS}(I, O)$ then every global test case for s is in $\mathcal{IOTS}(I, O \cup \{\delta\})$. As usual, in each state a test case must be able to accept any output from the SUT. Given I and O, the simplest test case is thus a process, that will be called $\perp$, which cannot send input to the SUT and thus whose traces are all elements of $(O \cup \{\delta\})^*$. We let $\perp_p$ denote the null process for port p, whose set of traces is $(O_p \cup \{\delta\})^*$.

As usual, a *test case* is an IOTS with a finite set of states. A test case t is *deterministic* if for every sequence σ we have that t **after** σ contains at most one process and t does not have a state where there is more than one input it can send to the SUT. It is normal to require that test cases are deterministic. In the distributed test architecture we place a local tester at each port and the tester at port p only observes the behaviour at p. We therefore require that a local test case contains a process for each port rather than a single process.

Definition 5. *Let $s \in \mathcal{IOTS}(I, O)$ be an IOTS with ports U and L. A* local test case *is a pair (t_U, t_L) of* local testers *in which:*

1. *t_U is a deterministic test case with input set I_U and output set $O_U \cup \{\delta\}$ and so is in $\mathcal{IOTS}(I_U, O_U \cup \{\delta\})$. In addition, if $t_U \stackrel{\sigma}{\Longrightarrow} t'_U$ for some σ then t'_U must be able to accept any value from $O_U \cup \{\delta\}$.*
2. *t_L is a deterministic test case with input set I_L and output set $O_L \cup \{\delta\}$ and so is in $\mathcal{IOTS}(I_L, O_L \cup \{\delta\})$. In addition, if $t_L \stackrel{\sigma}{\Longrightarrow} t'_L$ for some σ then t'_L must be able to accept any value from $O_L \cup \{\delta\}$.*

Since observations are made locally at each port it is natural to give each of the processes t_U and t_L a *pass* state and a $fail$ state. However, this approach leads to a loss of precision as we may observe traces at U and L that are individually consistent with s but where, when the logs of these traces are brought together, we know that there has been a failure. We know that there has been a failure if no interleaving of the behaviours observed at the two ports is consistent with the specification s. Let us consider, for example, a process s in which input of $?i_U$ at U is either followed by $!O_U$ at U and then output $!O_L$ at L or by $!O'_U$ at U and then output $!O'_L$ at L. If we use a local test case that applies $?i_U$, the sequence $?i_U!O_U$ is observed at U and $!O'_L$ is observed at L then each local tester observes behaviour that is consistent with projections of traces of the specification s and yet the pair of traces is not consistent with any conforming implementation.

It might seem sensible to represent pass and fail in terms of states of the pair of local testers and have unique *pass* and $fail$ states of this. However, an additional complication arises: Test effectiveness is not *monotonic*, that is, the

application of an input sequence can reveal a failure but we could extend this input sequence in a manner so that we lose the ability to find a failure. To see this, let us consider the following situation:

1. A specification s that has $?i_U$ at U then it is possible to apply $?i_U$ again and this is followed by output $!O_L$ at L.
2. An implementation i that has $?i_U$ at U, followed by $!O_L$ at L and then it is possible to apply $?i_U$.

If the local tester at U applies $?i_U$ and then observes output, and the local tester at L just observes output then we observe a failure since i will produce $!O_L$ at L when it should not have. However, if the local tester at U applies $?i_U?i_U$ and then observes output, and the local tester at L just observes output then we do not observe a failure since testing will observe $?i_U?i_U$ at U as expected and $!O_L$ at L as expected. As a result, in order to define verdicts we will need to define a mapping from states of a local test case to verdicts. In order to simplify the exposition we avoid this complexity and use the specification s as an *oracle*, as explained below in Definition 7.

It is important to consider what observations local test case (t_U, t_L) can make. As usual, we assume that quiescence can be observed and so (t_U, t_L) can observe the SUT being quiescent.[3] A test run for (t_U, t_L) is a sequence of observations that can occur in testing the SUT i with (t_U, t_L), that is, any sequence of observations that can be made with $t_U||i||t_L$. As usual, we require that testing is guaranteed to terminate in finite time.

Before we formally define the concept of a test run, we introduce new notation regarding how t_U, i and t_L can change through sending messages to one another.

Definition 6. *Let i be a SUT and (t_U, t_L) be a test case. We define the transitions $\xrightarrow{a}_t$ and $\xRightarrow{\sigma}_t$ as follows:*

1. *For $a \in \mathcal{A}ct$, if $i \xrightarrow{a} i'$ and $t_U \xrightarrow{a} t'_U$ then $t_U||i||t_L \xrightarrow{a}_t t'_U||i'||t_L$.*
2. *For $a \in \mathcal{A}ct$, if $i \xrightarrow{a} i'$ and $t_L \xrightarrow{a} t'_L$ then $t_U||i||t_L \xrightarrow{a}_t t_U||i'||t'_L$.*
3. *For $a = (!O_U, !O_L) \in O_U \times O_L$, if $i \xrightarrow{a} i'$, $t_U \xrightarrow{!O_U} t'_U$, and $t_L \xrightarrow{!O_L} t'_L$ then $t_U||i||t_L \xrightarrow{a}_t t'_U||i'||t'_L$.*
4. *If $i \xrightarrow{\tau} i'$ then $t_U||i||t_L \xrightarrow{\tau}_t t_U||i'||t_L$.*
5. *If $t_U||i||t_L \xrightarrow{a}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xrightarrow{a}_t$.*
6. *If $\sigma = a_1 \dots a_k \in (\mathcal{A}ct \cup \{\tau\})^*$, $t^1_U = t_U$, $i^1 = i$, $t^1_L = t_L$, for $1 \le j \le k$ we have $t^j_U||i^j||t^j_L \xrightarrow{a_j}_t t^{j+1}_U||i^{j+1}||t^{j+1}_L$, $t^{k+1}_U = t'_U$, $i^{k+1} = i'$, $t^{k+1}_L = t'_L$, then $t_U||i||t_L \xrightarrow{\sigma}_t t'_U||i'||t'_L$.*
7. *If $t_U||i||t_L \xrightarrow{\sigma}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xrightarrow{\sigma}_t$.*
8. *If $\sigma = a_1 \dots a_k \in \mathcal{A}ct^*$, $\sigma_1 \in \tau^* a_1 \tau^* \dots \tau^* a_k \tau^*$ and $t_U||i||t_L \xrightarrow{\sigma_1}_t t'_U||i'||t'_L$ then we can write $t_U||i||t_L \xRightarrow{\sigma}_t t'_U||i'||t'_L$.*

---

[3] While the observation of quiescence in the distributed test architecture is slightly different, since it is possible to have inactivity at one port when there is activity at another, in practice there are few differences since in testing we know the test case being applied.

9. *If* $t_U||i||t_L \stackrel{\sigma}{\Longrightarrow}_t t'_U||i'||t'_L$ *then we can write* $t_U||i||t_L \stackrel{\sigma}{\Longrightarrow}_t$ *.*

We are now in a position to define what it means for an implementation to pass a test run with a local test case and thus to pass a local test case. We require a test run to end with i being quiescent. In order to see why we require this consider s_1 and i_1 shown in Figure 2. It is clear that i is a good implementation of s when considering the distributed test architecture. Nevertheless, if we consider the trace $?i_U!O_L$ of i_1 we find that this is not a permutation of a trace of s. However, testing cannot observe $?i_U!O_L$. This differs from the case where we observe global traces: if we can observe a global trace σ then we can construct the prefixes of σ. However, in general this cannot be done when testing in the distributed test architecture.

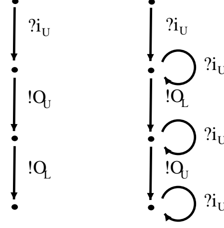

Fig. 2. Processes s_1 and i_1

The following defines test runs with local test cases and what it means for the SUT to pass a test run and to pass a local test case.

Definition 7. *Let* s *be a specification,* i *be a SUT and* (t_U, t_L) *be a local test case. We introduce the following notation.*

1. *A trace* σ *is a* test run *for* i *with local test case* (t_U, t_L) *if there exists* t'_U*,* t'_L*, and* i' *such that* $t_U||i||t_L \stackrel{\sigma}{\Longrightarrow}_t t'_U||i'||t'_L$ *and* $t'_U||i'||t'_L$ *is quiescent.*
2. *Implementation* i passes *a test run* σ *with* (t_U, t_L) *for* s *if either of the following two conditions hold.*
   (a) *There does not exist a trace* σ' *of* s *such that* $in(\sigma) \sim in(\sigma')$*.*
   (b) *there exists some* $\sigma' \sim \sigma$ *that is a trace of* s*.*
3. *If* i *passes the test run* σ*, and* s *can be inferred from the context, we write* $t_U||i||t_L \stackrel{\sigma}{\Longrightarrow} pass$*. Otherwise we say* i fails *the test run* σ *and we write* $t_U||i||t_L \stackrel{\sigma}{\Longrightarrow} fail$*.*
4. *Implementation* i passes *test case* (t_U, t_L) *for* s *if* i *passes every possible test run of* i *with* (t_U, t_L) *for* s *and this is denoted* $PASSES(i, t_U, t_L, s)$*. Otherwise* i fails (t_U, t_L) *and this is denoted* $FAILS(i, t_U, t_L, s)$*.*

The first clause of the second item corresponds to σ not being in response to an input sequence $in(\sigma)$ such that the behaviour of s is defined for an input sequence that is indistinguishable from $in(\sigma)$ in the distributed test architecture. Moreover, while the second clause of the same item does not say that s must be quiescent after σ' this is implicit when i passes a local test case (t_U, t_L) since if i is quiescent after σ then i is quiescent after $\sigma\delta$ and so this is also a possible test run for i with (t_U, t_L).

Even if we bring together the logs from the testers at the end of a test run, the use of the distributed test architecture can reduce the ability of testing to distinguish between the specification s and the SUT i.

Proposition 1. *It is possible for an implementation to fail a test case if global traces are observed and yet pass the test case if only local traces are observed.*

Proof. Consider the specification s_1 and the implementation i_1 illustrated in Figure 2. Here the only sequence of actions in s_1 is $?i_U$ at U followed by $!O_U$ at U, then $!O_L$ at L. The implementation i has $?i_U$ at U followed by $!O_L$ at L, then $!O_U$ at U. So, clearly i_1 does not conform to s_1 under the **ioco** relation and will fail a global test case that applies $?i_U$. However, it is not possible for the local testers to observe the difference if we apply $?i_U$ since the tester at U will see the expected sequence of actions $?i_U!O_U$ and the tester at L will observe the expected sequence of actions $!O_L$.

4 A new implementation relation

We have seen that an implementation may pass a test case when we only make local observations and yet fail the test case if observations are made globally. This suggest that the implementation relation required when making local observations will differ from the implementation relation **ioco** used when making global observations. In this section we define a new implementation relation.

It might seem natural to define the new implementation relation so that the behaviour of the implementation i at a port p must conform to the behaviour of s at port p and this must hold for every port. However, we have already seen that there could be a trace σ such that $i \stackrel{\sigma}{\Longrightarrow}$, the projections of σ at ports U and L are individually consistent with traces of s and yet there is no $\sigma' \sim \sigma$ that is a trace of s. As a result, we require an implementation relation that compares pairs of local traces with global traces of the specification.

We want the new implementation relation, that we call **dioco**, to correspond to the ability of testing to determine when an implementation conforms to a given specification when testing in the distributed test architecture. We can define this implementation relation in terms of the traces in the implementation.

Definition 8. *Given specification s and implementation i we have that i* **dioco** *s if for every trace σ such that $i \stackrel{\sigma}{\Longrightarrow} i'$ for some i' that is quiescent, if there is a trace σ_1 of s such that $in(\sigma_1) \sim in(\sigma)$ then there exists a trace σ' such that $s \stackrel{\sigma'}{\Longrightarrow} s'$ and $\sigma' \sim \sigma$.*

The implementation relation **dioco** captures our notion of test run and failing a test run with a local test case.

Proposition 2. *Given a specification s and an implementation i, i* **dioco** *s if and only if for every local test case (t_U, t_L) we have that $PASSES(i, t_U, t_L, s)$.*

Proof. First let us assume that i **dioco** s and let (t_U, t_L) be a local test case. We require to prove that i passes all possible test runs with (t_U, t_L) for s. Let σ denote some trace that can be produced by a test run of i with (t_U, t_L). If there does not exist a trace σ' of s such that $in(\sigma) \sim in(\sigma')$ then i passes this test run by definition. We therefore assume that there is some such σ'. But, since i **dioco** s we must have that there is a trace σ' of s such that $\sigma' \sim \sigma$ and so i passes this test run with (t_U, t_L). It thus follows that i passes all possible test runs with (t_U, t_L) for s, as required.

Now let us assume that i passes all possible test runs and let σ be a trace such that $i \stackrel{\sigma}{\Longrightarrow} i'$ for some i' that is quiescent. Thus, we consider the situation in which there exists a trace σ_1 of s such that $in(\sigma_1) \sim in(\sigma)$ and we are required to prove that there exists some σ' such that $s \stackrel{\sigma'}{\Longrightarrow}$ and $\sigma' \sim \sigma$. Let $t_p = \pi_p(\sigma)$ for $p \in \{U, L\}$. Then clearly σ is a possible test run of i with local test case (t_U, t_L) and so the result follows from the fact that i must pass all possible test runs with (t_U, t_L).

The next result shows that the implementation relations **dioco** and **ioco** are incomparable.

Proposition 3. *There exist processes s and i such that i* **dioco** *s but not i* **ioco** *s. There also exist processes s and i such that i* **ioco** *s but not i* **dioco** *s.*

Proof. Consider the processes s_1 and i_1 given in Figure 2. It is clear that i_1 **dioco** s_1 since the local testers cannot distinguish between the traces $?i_U!O_U!O_L$ and $?i_U!O_L!O_U$. However, it is also clear that $out(i_1 \textbf{ after } ?i_U) \not\subseteq out(s_1 \textbf{ after } ?i_U)$ and so we do not have that i_1 **ioco** s_1.

Now consider processes s_2 and i_2 given in Figure 3. The only traces that i_2 and s_2 have in common are those in the sets δ^*, $\delta^*?i_L\delta^*$, and $\delta^*?i_L\delta^*?i_U$ δ^* and for each of these i_2 and s_2 have the same set of possible outputs, δ^*. Thus, i_2 **ioco** s_2. However, local testers cannot distinguish between $?i_L?i_U$ and $?i_U?i_L$. We have that the second of these sequences leads to an output of $!O_U$ in i_2 and this is not allowed after $?i_L?i_U$ in s_2. As a result we do not have that i_2 **dioco** s_2, as required.

The use of the distributed test architecture reduces the ability of testing to observe behaviours of the SUT and thus it is natural to expect that there are cases where the resultant implementation relation allows an implementation i to conform to s even though we do not have that i **ioco** s. However, one would expect **dioco** to be strictly weaker than **ioco** since all observations that can be made locally can also be made globally. The example presented in Figure 3, and used in Proposition 3, shows that this is not the case. This example relies on s

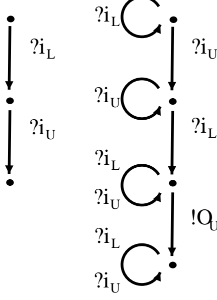

Fig. 3. Processes s_2 and i_2

not being input enabled: We do not have i **dioco** s because the 'problematic' behaviour in i_2 occurs after a trace σ that is not in s_2 but such that there is a permutation $\sigma' \sim \sigma$ that is in s_2. The following results show that if the specification and implementation are input enabled then we obtain the expected result: **dioco** is strictly weaker than **ioco**.

Proposition 4. *If s and i are input enabled and i **ioco** s then every trace of i is also a trace of s.*

Proposition 5. *If s and i are input enabled then whenever we have that i **ioco** s it must be the case that i **dioco** s.*

Proof. Assume that i **ioco** s, $i \stackrel{\sigma}{\Longrightarrow} i'$ for some i' that is quiescent, and there is a trace σ_1 of s such that $in(\sigma_1) \sim in(\sigma)$. It is sufficient to prove that there is a trace σ' of s such that $\sigma' \sim \sigma$. This follows immediately from the fact that, by Proposition 4, σ is a trace of s.

While for input enabled specification s and implementation i we have that i **ioco** s implies i **dioco** s, the converse does not hold. In order to see this, consider the processes in Figure 4. It is clear that these are incomparable under **ioco** since if an output is produced then the value of this is determined by the initial input and the mapping from initial input to output is different in the two cases. However, in each case the two paths to the states from which there can be output simply require that there is at least one use of $?i_U$ and at least one use of $?i_L$. As a result, the sets of paths to the two states from which output can be produced are indistinguishable when observing events locally and so the two processes cannot be distinguished when testing in the distributed test architecture.

It is usual to insist that there is no 'unnecessary nondeterminism' and thus the tester cannot reach a state in which it could provide alternative inputs to the implementation (see, for example, [9]). This ensures that the test is controllable. In the next section we interpret this in the context of the distributed test architecture.

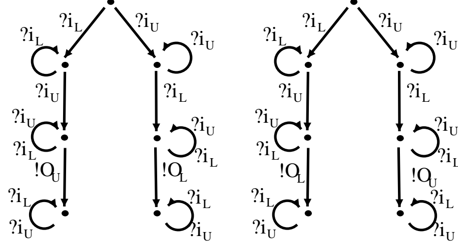

Fig. 4. Processes that are not related under **ioco**

5 Deterministic test cases

Since test objectives are usually stated at the specification level it is natural to initially generate a global test case that achieves a given test objective. However, there is then the challenge of producing local testers that implement such a global test case. In this section we assume that a deterministic global test case t has been produced for testing an implementation i against specification s.

It is normal to use deterministic test cases and so for (t_U, t_L) we already assumed in Definition 5 that t_U and t_L are deterministic. However, it is possible that (t_U, t_L) is nondeterministic despite t_U and t_L being deterministic. A simple example of this is any pair (t_U, t_L) in which t_U and t_L both start by sending an input to the implementation: Even if t_U and t_L are deterministic, the order in which the implementation receives these inputs from t_U and t_L is not predictable. Nondeterminism occurs in (t_U, t_L) if both t_U and t_L can provide an input to the implementation at some point in a test run. In this section we define what it means for local test case (t_U, t_L) to be deterministic for a given specification s and show how we can map a global test case to a local test case.

It might seem desirable that a local test case is deterministic for any possible implementation. However, in Proposition 6 we will prove that for an input enabled specification if the interaction of a deterministic local test case (t_U, t_L) with an implementation i can lead to nondeterminism then there is a local test case (t'_U, t'_L) that i fails and shows how (t'_U, t'_L) can be constructed from (t_U, t_L). Thus, it is only necessary to consider traces that can be produced by the specification and test case interacting. Let us note that since the testers are local, this is equivalent to considering all traces that can be produced by the local test case combined with the specification and all permutations that preserve the traces at each port.

It is interesting to note that for any local test case (t_U, t_L) such that t_U and t_L both have the potential to send input to the SUT, there is some behaviour of a possible implementation that will lead to nondeterminism. This is the case since there must exist a possible trace σ_U at U after which t_U will send input to the SUT and a possible trace σ_L at L after which t_L will send input to the

SUT and so there is nondeterminism if the SUT produces any trace $\sigma' \sim \sigma_U \sigma_L$ in testing. As a result, it is unreasonable to require more than for the local test case to be deterministic for all possible behaviours of the specification.

Definition 9. *Given a specification s we say that the local test case (t_U, t_L) is* deterministic for *s if there do not exist traces σ_1 and σ_2, with $\sigma_2 \sim \sigma_1$, and $a_1, a_2 \in I$, with $a_1 \neq a_2$, such that $t_U||s||t_L \stackrel{\sigma_1 a_1}{\Longrightarrow}_t$ and $t_U||s||t_L \stackrel{\sigma_2 a_2}{\Longrightarrow}_t$. Given a specification s, we let $\mathcal{T}_D(s)$ denote the set of local test cases that are deterministic for s.*

Interestingly, this is similar to the notion of local choice defined in the context of MSCs [19]. We can now prove that given local test case (t_U, t_L) that is deterministic for specification s, the application of (t_U, t_L) to an implementation i can only be nondeterministic through an erroneous behaviour of i and there is a prefix of (t_U, t_L) that is capable of detecting this erroneous behaviour through local observations. The proof will use the notion of a prefix of a local test case (t_U, t_L), which is a local test case that can be generated from (t_U, t_L) by replacing one or more of the input states[4] of t_U and t_L by processes that can only receive output.

Proposition 6. *Let us suppose that s is an input enabled specification, i is an implementation, and $(t_U, t_L) \in \mathcal{T}_D(s)$ is a deterministic test case for s. If there exists σ_1 and σ_2, with $\sigma_2 \sim \sigma_1$, and $a_1, a_2 \in I$, with $a_1 \neq a_2$, such that $t_U||i||t_L \stackrel{\sigma_1}{\Longrightarrow}_t t'_U||i'||t'_L$, $t_U||i||t_L \stackrel{\sigma_2}{\Longrightarrow}_t t''_U||i''||t''_L$, $t'_U||i'||t'_L \stackrel{a_1}{\Longrightarrow}$, and $t''_U||i''||t''_L \stackrel{a_2}{\Longrightarrow}$ then there exists a prefix (t^1_U, t^1_L) of (t_U, t_L) such that i fails (t^1_U, t^1_L).*

Proof. By definition, since (t_U, t_L) is deterministic for s, $\sigma_1 \sim \sigma_2$ can be followed by different inputs when applying (t_U, t_L), we must have that $\neg(s \stackrel{\sigma'}{\Longrightarrow})$ for all $\sigma' \sim \sigma_1$. Let σ be the shortest prefix of σ_1 such that $t_U||i||t_L \stackrel{\sigma}{\Longrightarrow}_t t^1_U||i^1||t^1_L$ for some i^1 that is quiescent after σ and no $\sigma'_1 \sim \sigma$ is a trace of s. If $t^1_U||i^1||t^1_L$ cannot perform any actions then σ is a trace that can be produced by a test run of i with (t_U, t_L) and so the result follows from the fact that s is input enabled. Otherwise we have that (t^1_U, t^1_L) is in an input state and so we can define a prefix (t''_U, t''_L) of (t_U, t_L) by replacing the processes t^1_U and t^1_L, of t_U and t_L, by the processes $\perp_U$ and $\perp_L$, respectively, that cannot send input to the SUT. Then the interaction of i with (t''_U, t''_L) can lead to testing terminating with the trace σ and thus at least one tester observing a failure.

Given local test case (t_U, t_L), it is possible to produce the set of prefixes of (t_U, t_L) and use these in testing. If we do this then an implementation that passes all of these local test cases does not lead to nondeterminism when tested with (t_U, t_L). If we have a deterministic global test case t, we want to devise local testers t_U and t_L that implement t.

---

[4] An input state of process s is a reachable state r that can perform an input, that is, there exists σ such that $s \stackrel{\sigma}{\Longrightarrow} r$ and an input $?i$ such that $r \stackrel{?i}{\longrightarrow}$.

Definition 10. *A local test case* (t_U, t_L) implements *the global test case* t *for specification* s *if we have that the interaction between* (t_U, t_L) *and* s *can lead to trace* σ *if and only if the interaction between* t *and* s *can produce trace* σ*. More formally, for every trace* σ *we have that* $t_U||s||t_L \stackrel{\sigma}{\Longrightarrow}$ *if and only if* $t||s \stackrel{\sigma}{\Longrightarrow}$*.*

We can produce local testers by taking projections of a global test case t. In taking the projection of t at p, we will eliminate actions at port $q \neq p$.

Definition 11. *Given global test case* t *and port* p*, we let* $local_p(t)$ *denote the local tester at* p *defined by the following rules.*

1. *If* t *is the null process* $\perp$*, that cannot send an input, then* $local_p(t)$ *is* $\perp_p$*.*
2. *If* $a \in I_p \cup O_p \cup \{\delta\}$ *then* $local_p(a.s) = a.local_p(s)$
3. *If* $a \in I_q \cup O_q$*,* $q \neq p$*, then* $local_p(a.s) = local_p(s)$
4. *If* $a = (!O_U, !O_L)$*,* $!O_p \neq -$*, then* $local_p(a.s) = !O_p.local_p(s)$
5. *If* $a = (!O_U, !O_L)$*,* $!O_p = -$*, then* $local_p(a.s) = local_p(s)$
6. $local_p(t_1 + \ldots + t_k) = local_p(t_1) + \ldots + local_p(t_k)$*.*

If we give the function $local_p$ a deterministic global test case then it returns the local testers we require.

Proposition 7. *Given deterministic global test case* t *and* $t_p = local_p(t)$ *for* $p \in \{U, L\}$ *we have that if the local test case* (t_U, t_L) *is deterministic then* (t_U, t_L) *implements* t*.*

Proof. We prove the result by induction on the size of t, which is the number of nodes of the tree formed from t. Clearly the result holds for the base case where t has only one node and so is the null test case $\perp$. Inductive hypothesis: We assume that the result holds for every global test case of size less than k, for $k > 1$, and t has size k. There are three cases to consider.

Case 1: t starts by sending input $?i$ to the SUT, that is, $t \xrightarrow{?i} t'$ for some t'. Since t is deterministic, t' is uniquely defined. Without loss of generality, $?i$ is at port U and $t_U \xrightarrow{?i} t'_U$ (the proof for $i?$ being at port L is similar). Let S be the set of processes that s can become after $?i$, that is, $S = \{s_1 | s \xrightarrow{?i} s_1\}$, and let us consider $s' = \sum S$. Clearly, t' and (t'_U, t_L) are deterministic for s' and the size of t' is less than k. Further, $t_L = local_L(t')$ and $t'_U = local_U(t')$ and so, by the inductive hypothesis, we have that a trace σ' can be produced by the interaction between t' and s' if and only if it can be produced by the interaction between (t'_U, t_L) and s'. A trace σ can be produced by the interaction between t and s if and only if σ is $?i\sigma'$ for some trace σ' that can be produced by an interaction between s' and t'. Since (t_U, t_L) is deterministic for s we also have that a trace σ can be produced by the interaction between (t_U, t_L) and s if and only if σ is $?i\sigma'$ for some trace σ' that can be produced by an interaction between s' and (t'_U, t_L). The result thus follows.

Case 2: t starts with an output from the SUT, possibly branching on different outputs. For each output y that s can produce from its initial state we let t^y, t^y_U, t^y_L, and s^y be defined by $t \xrightarrow{y} t^y$ and $t_U||s||t_L \xrightarrow{y}_t t^y_U||s^y||t^y_L$. Thus, $s^y = \sum S$,

being S the set of processes that s can become after y. Let $y_p = y|_p$ for $p \in \{U, L\}$ and so $t_p \xrightarrow{y_p} t_p^y$. Clearly t^y and (t_U^y, t_L^y) are deterministic for s^y. We can now apply the inductive hypothesis to t^y, (t_U^y, t_L^y), and s^y and the result follows.

Case 3: t starts with δ and so t_U and t_L both start with δ. Then $s \stackrel{\delta}{\Longrightarrow} s$ and so the result follows from the inductive hypothesis.

6 Conclusions

This paper has investigated testing from an input output transition system in the distributed test architecture. The problem of testing from a deterministic finite state machine in this architecture has received much attention but, while deterministic finite state machines are appropriate for modelling several important classes of system, input output transition systems are more general.

When testing in the distributed test architecture each tester only observes the actions at its port. As a result, it is not possible to reconstruct the global trace after testing and this has an effect on the ability of testing to distinguish between two processes. We introduced a new implementation relation **dioco** that captures the ability of testing to compare an implementation and a specification: i **dioco** s if and only if it is possible for testing to show that i does not correctly implement s when testing in the distributed test architecture.

It is normal to require a test case to be deterministic. However, there are test cases that are deterministic if there is a single global tester that interacts with the SUT at all its ports but that cannot be implemented as a deterministic local test case when testing in the distributed test architecture. We have defined a function that takes a global test case and returns a local test case that consists of a tester for each port. We have proved that this function returns a local test case if and only if it is possible to implement the global test case using a deterministic local test case.

There are several avenues of future work. First, we could parameterize the implementation relation with a set of input sequences or traces. In addition, there is the problem of automatically generating test cases that can be implemented as deterministic local test cases. It has been shown that controllability and observability problems, when testing from a deterministic finite state machine, can be overcome if the testers can exchange coordination messages through an external network and there should be scope for using coordination messages when testing from an input output transition system. Finally, recent work has described systems in which an operation is triggered by receiving input at more than one port [20] and it would be interesting to extend the work to such systems.

References

1. ISO/IEC JTC 1, J.T.C.: International Standard ISO/IEC 9646-1. Information Technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts. ISO/IEC (1994)

2. Sarikaya, B., Bochmann, G.v.: Synchronization and specification issues in protocol testing. IEEE Transactions on Communications **32** (1984) 389–395
3. Luo, G., Dssouli, R., Bochmann, G.v.: Generating synchronizable test sequences based on finite state machine with distributed ports. In: 6th IFIP Workshop on Protocol Test Systems, IWPTS'93, North-Holland (1993) 139–153
4. Tai, K.C., Young, Y.C.: Synchronizable test sequences of finite state machines. Computer Networks and ISDN Systems **30**(12) (1998) 1111–1134
5. Rafiq, O., Cacciari, L.: Coordination algorithm for distributed testing. The Journal of Supercomputing **24**(2) (2003) 203–211
6. Ural, H., Williams, C.: Constructing checking sequences for distributed testing. Formal Aspects of Computing **18**(1) (2006) 84–101
7. Khoumsi, A.: A temporal approach for testing distributed systems. IEEE Transactions on Software Engineering **28**(11) (2002) 1085–1103
8. Hierons, R.M., Ural, H.: The effect of the distributed test architecture on the power of testing. The Computer Journal **51**(4) (2008) 497–510
9. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software – Concepts and Tools **17**(3) (1996) 103–120
10. Tretmans, J.: Testing concurrent systems: A formal approach. In: 10th Int. Conf. on Concurrency Theory, CONCUR'99, LNCS 1664, Springer (1999) 46–65
11. Núñez, M., Rodríguez, I.: Towards testing stochastic timed systems. In: 23rd IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'03, LNCS 2767, Springer (2003) 335–350
12. Brandán Briones, L., Brinksma, E.: A test generation framework for quiescent real-time systems. In: 4th Int. Workshop on Formal Approaches to Testing of Software, FATES'04, LNCS 3395, Springer (2004) 64–78
13. Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: 11th Int. SPIN Workshop on Model Checking of Software, SPIN'04, LNCS 2989, Springer (2004) 109–126
14. Bijl, M.v., Rensink, A., Tretmans, J.: Action refinement in conformance testing. In: 17th Int. Conf. on Testing of Communicating Systems, TestCom'05, LNCS 3502, Springer (2005) 81–96
15. López, N., Núñez, M., Rodríguez, I.: Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science **353**(1–3) (2006) 228–248
16. Frantzen, L., Tretmans, J., Willemse, T.: A symbolic framework for model-based testing. In: 1st Combined Int. Workshops on Formal Approaches to Software Testing and Runtime Verification, FATES 2006 and RV 2006, LNCS 4262, Springer (2006) 40–54
17. Merayo, M.G., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer Networks **52**(2) (2008) 432–460
18. Brinksma, E., Heerink, L., Tretmans, J.: Factorized test generation for multi-input/output transition systems. In: 11th IFIP Workshop on Testing of Communicating Systems, IWTCS'98, Kluwer Academic Publishers (1998) 67–82
19. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Transactions on Software Engineering **29**(7) (2003) 623–633
20. Haar, S., Jard, C., Jourdan, G.V.: Testing input/output partial order automata. In: Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'07, and 7th Int. Workshop on Formal Approaches to Software Testing, FATES'07, LNCS 4581, Springer (2007) 171–185