# Implementation Relations for Distributed Testing

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

[1] Department of Computer Science, The University of Sheffield, Sheffield, S1 4GG, United Kingdom
[2] Design and Testing of Reliable Systems Research Group Universidad Complutense de Madrid, Madrid, Spain

**Abstract.** When testing a system that interacts with its environment at several physically distributed interfaces (ports) it is normal to place a local tester at each port. If the local testers do not synchronise their actions then the local tester at port $p$ can only observe the sequence of inputs and outputs that occur at $p$. If, in addition, there is no global clock then it may be impossible to reconstruct the global trace that occurred in testing and testing is then using the distributed test architecture. As a result, the System Under Test (SUT) might be able to produce a global trace that is not allowed by the specification, and so would normally represent a failure, but where the local testers cannot observe this difference. The use of the distributed test architecture thus affects the ability of testing to distinguish between a specification and an SUT and so leads to the need for a different notion of correctness (implementation relation). This paper explores alternative implementation relations for distributed testing and how they relate.

## 1 Introduction

Jan Peleska has made a significant long-term contribution to the development of systematic test generation techniques based on formal models (see, for example, [19]) and has shown how such techniques can be used in an industrial setting [24,25,26]. This is an important contribution since testing is a core part of software development. As Peleska has shown, if there is a model of the required behaviour of the *system under test (SUT)* then there is potential to automate test generation based on this model, with this approach often being called *model-based testing (MBT)*. Further, if the model has a formal semantics then automated test generation can be systematic, in the sense that one can formally reason about the types of faults that test cases can find (see, for example, [1,7,19,20,24,27,28,30]).

Most work on MBT uses models in the form of a finite state machine (FSM) or labelled transition system (LTS). However, the user is not expected to produce FSM or LTS models: the user can produce models written using a state-based language such as Statecharts, with these models being mapped to FSMs or LTSs [6,7,20]. Testing is typically then a process in which a tester interacts

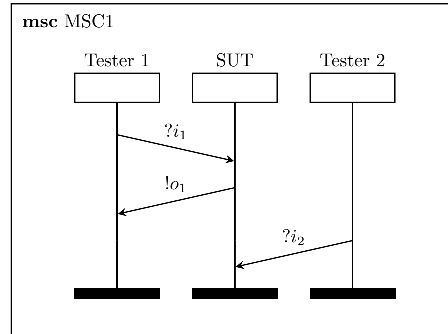


**Fig. 1.** A controllability problem [15]

with the SUT, through providing inputs and observing outputs, and the resultant sequence of inputs and outputs (*trace*) is checked against the original model/specification. Although this captures how testing is often carried out, testing can be rather different. For example, the communication between the tester and the SUT might be through a medium that introduces a delay. Testing is then asynchronous, with the trace that is observed by the tester potentially not being the trace produced by the SUT since the tester observes inputs before the SUT does and the SUT produces outputs before they are observed by the tester [9,12,32]. The SUT might also interact with its environment at multiple physically distributed interfaces, called *ports*, with there being a *local tester* at each port. If the local testers do not synchronise their actions and there is no global clock then testing is taking place in the distributed test architecture [21]. We use the term *distributed testing* when we refer to testing in the distributed test architecture.

In the distributed test architecture, an observation consists of a number of *local traces*, one for each port, as opposed to a single (global) trace. Early work on distributed testing noted that it can lead to *controllability problems*, which occur because a tester cannot observe the interactions at other ports and, therefore, sometimes does not know when to supply an input [4,29]. To see how controllability problems can occur, consider the scenario shown in Figure 1 in which three processes interact (two testers and the SUT), arrows represent the exchange of messages, and time progresses as we move down the line associated with a process. Here, Tester 1 starts by sending input $?i_1$ to the SUT and should then receive output $!o_1$. After this, Tester 2 should send input $?i_2$. However Tester 2 cannot observe the interactions between Tester 1 and the SUT and so does not know when to send its input.

There can also be *observability problems*, where a global trace not allowed by the specification occurs but the observation made (the set of local traces) is consistent with a behaviour of the specification [5]. To see how observability problems can occur, consider the two scenarios shown in Figure 2. Here, there

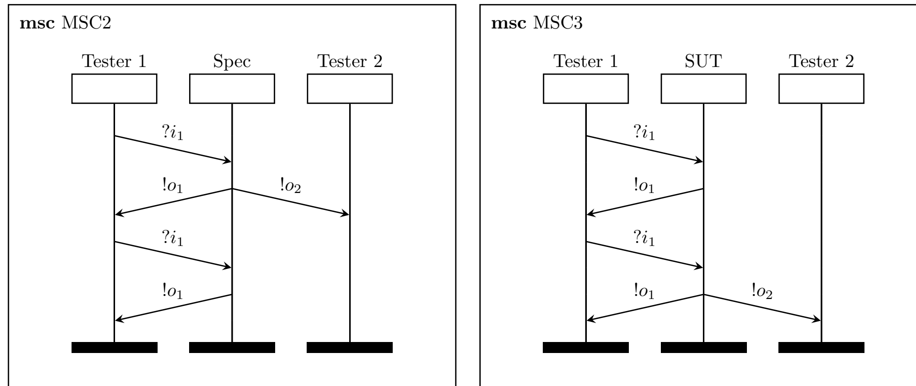


**Fig. 2.** Observationally equivalent scenarios [15]

are two different global traces but in each case Tester 1 observes $?i_1!o_1?i_1!o_1$ and Tester 2 observes $!o_2$. As a result, these global traces are indistinguishable when testing in the distributed test architecture.

Much of the early work on distributed testing aimed to produce test generation techniques that returned test sequences that do not suffer from controllability or observability problems. This previous work thus used traditional implementation relations such as **ioco** [30]: the implementation relation did not reflect the reduced ability of testing to distinguish between different global traces and so also different processes. As a result, for example, such work might consider a test case to be sufficient to find a given fault even when no tester can observe a difference in behaviour (again, see Figure 2). This paper focuses on later work that developed new implementation relations that reflected the nature of the distributed test architecture and the ability of testing to distinguish processes in this test architecture.

This paper is structured as follows. Section 2 defines the types of models considered and introduces notation used throughout the paper. Section 3 then formalises what we mean by distributed testing and Section 4 defines and compares the implementation relations. Section 5 then outlines some related and future work and Section 6 draws conclusions.

## 2 Preliminaries

In software testing, typically a tester applies inputs and observes outputs produced by the SUT. Throughout this paper we use $I$ to denote the set of possible inputs and $O$ to denote the set of possible outputs. The sets $I$ and $O$ are therefore disjoint. We will normally precede the name of an input by '?' and the name of an output by '!'. We will use a running example to illustrate the key principles.

*Example 1.* The system depicted in Figure 3 represents a simplified version of the diagnosis protocol of a gynaecological cancer screening centre management system. It focuses on the functionality associated with the process that begins at the moment a patient makes a date with the doctor. When a patient visits the doctor, they can either prescribe some tests or diagnose an illness. In the first case, the patient must go to the laboratory and image diagnosis section and make the corresponding appointments. Once the results of the tests are available, the patient will visit the doctor. If the results of the tests provide enough information, then the doctor will diagnose the patient and prescribe the appropriate medication. However, the doctor may need more tests to give a final diagnosis and then the patient will begin the cycle again. The protocol is very close to a *real* system. In order to simplify the presentation we only consider one battery of tests: an ultrasound, a mammography and a smear test. After the test results are received in the doctor's office and the patient makes an appointment, the patient will visit the doctor for a diagnosis.

The main type of model we use is an input output transition system, which is a labelled transition system in which the set of actions is partitioned into inputs and outputs.

**Definition 1 (Input Output Transition System).** *An input output transition system (IOTS) $r$ is defined by a tuple $(Q, I, O, T, q_{in})$ 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 $T \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$ is the transition relation. Here, $\tau$ represents an internal action, which cannot be observed.*

*We say that state $q \in Q$ is* stable *if there is no $q' \in Q$ and $y \in O \cup \{\tau\}$ such that $(q, y, q') \in T$. This represents the situation in which $r$ cannot change state without first receiving input. The process $r$ 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 make the normal assumption that the SUT is input-enabled. In defining implementation relations for distributed testing, we will also require that specifications are input-enabled[3]. Some of the implementation relations described in this paper have been generalised to the case where the specification need not be input-enabled [17].

*Example 2.* The specification depicted in Figure 3 is an $IOTS$ in which input actions represent different actions, such as the request of the different appointments ($?app_smear_test$, $?app_ultrasound$, ...), the inclusion in the system of the images or samples obtained by means of tests ($?smear_test$, $?mammography$, ...), the registration of the diagnosis ($?diagnosis$) or the prescription of tests ($?tests_presc$). The output actions correspond to the information provided by the system to the users. For example, the dates of the requested appointments ($!date_smear_test$, $!date_ultrasound$, ...) or the results obtained from the tests

[3] The alternative term Input Output Labelled Transition System is often used if a process does not have to be input-enabled.

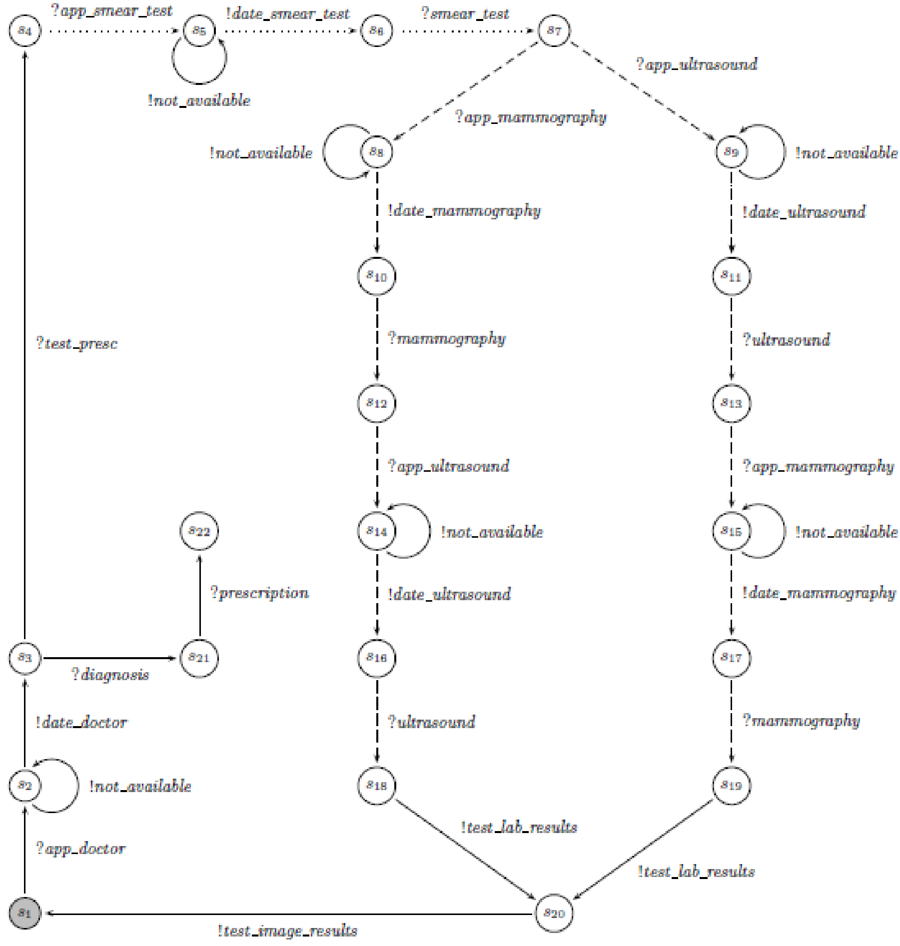


**Fig. 3.** Specification of the appointments protocol

carried out (!*test_lab_results*, !*test_image_results*). The initial state, $s_1$, is shaded. For the sake of clarity, not all transitions are included in the figure since this would overload the graph. Specifically, we have omitted those required to ensure that the system is input-enabled (the missing transitions lead to no change in state).

We also introduce notation that can be used to define processes. Given action $a$ and process $r$, we use $a.r$ to denote the process that becomes $r$ after engaging in action $a$. Further, if $S$ is a countable set of processes then we use $\sum S$ to denote the process that non-deterministically chooses to be any process in $S$.

As is usually done when testing from an IOTS, we assume that the tester can observe the SUT being in a stable state (being *quiescent*). In practice, the

tester will do this via a timeout, with the time $\Delta_T$ used being problem-specific. There is thus the associated test hypothesis (assumption) that if the SUT does not receive input or produce output for time $\Delta_T$ then the SUT is in a stable state. We use $\delta$ to denote quiescence.

**Definition 2.** *Given IOTS $r = (Q, I, O, T, q_{in})$, we can extend the transition relation $T$ to $T_\delta$ by adding the transition $(q, \delta, q)$ for each stable state $q$ of $r$. We use $\mathcal{A}ct$ to denote the set of observable actions and so $\mathcal{A}ct = I \cup O \cup \{\delta\}$.*

Note that traces that (can) include quiescence are often called *suspension traces*; we simply call them traces since we do not consider other types of traces. The following standard notation is often used in the context of the standard implementation relation **ioco** (see, for example, [30]).

**Definition 3.** *Let $r = (Q, I, O, T, q_{in})$ 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 $r \stackrel{\sigma}{\Longrightarrow}$ if there exists $q'$ such that $q_{in} \stackrel{\sigma}{\Longrightarrow} q'$ and we say that $\sigma$ is a* trace *of $r$.*
6. *We let $\mathcal{T}r^*(r)$ denote the set of finite traces of $r$.*

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

1. *$q$ **after** $\sigma = \{q' \in Q | q \stackrel{\sigma}{\Longrightarrow} q'\}$.*
2. *$r$ **after** $\sigma = q_{in}$ **after** $\sigma$.*
3. *$out(q) = \{!o \in O \cup \{\delta\} | q \stackrel{!o}{\Longrightarrow}\}$.*

*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 the process $r$ is* deterministic *if for every state $q$ and $a \in \mathcal{A}ct$ there is at most one state $q'$ such that $(q, a, q') \in T_\delta$. We say that $r$ is* output-divergent *if it can reach a state from which there is an infinite trace that contains outputs and internal actions only.*

We will consider processes that are output-divergent but a number of the definitions will require us to restrict attention to processes that are not output-divergent. Note that output divergence can be undesirable in testing since a process can choose to keep on providing outputs and not allow the tester to supply inputs. We can now define the standard implementation relation **ioco**.

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

## 3 Distributed Testing

Implementation relations such as **ioco** implicitly assume that there is a single *global tester* and this global tester is able to observe all of the actions in which the SUT engages, as well as quiescence, and determine the order in which these actions occurred. For example, if the SUT corresponding to the specification presented in Figure 3 produced output !*test_lab_results* and then !*test_image_results* then the tester can observe both outputs and know that they were produced in this order. For many systems, this is a reasonable assumption and so one can use this type of implementation relation.

Research in the 1980s, on testing implementations of communication protocols [4,5] against an FSM specification, observed that sometimes one requires multiple testers. In this work, there was an upper tester, which acted as the software that was using the protocol, and a lower tester, which interacted with the SUT through a communications network. For such systems, the SUT has multiple (two) physically distributed interfaces (called *ports*), there is a separate *local tester* at each port, and these local testers are not synchronised. This results in the local tester at port $p$ observing a local trace (the sequence of events at port $p$) and so the overall observation made being a set of local traces: one local trace for each port of the SUT.

Figure 4 shows two architectures that can be used when testing a system that has multiple ports [17]. Figure 4(a) shows the case where there is a single global tester that provides inputs at all of the ports and observes the outputs; this is consistent with implementation relations such as **ioco**. Such a global tester can reconstruct the global trace that occurred during testing, although sometimes it may be difficult for the global tester to achieve this if the observation of an event is given a local timestamp and the clocks used are not perfectly synchronised. Figure 4(b) shows the *distributed test architecture* in which there is a separate local tester at each port and each local tester observes a local trace. It is possible to combine the two test architectures, leading to there being both a centralised tester and local testers [13] but we will not discuss such a combined architecture.

As previously mentioned, early work on distributed testing observed that it can lead to additional controllability and observability problems. The initial response was to try to find test sequences that avoid controllability and observability problems. These are test sequences (traces) where, for example, one can establish a global order of actions (see, for example, [2,11,22,23,31]). Later, it was recognised that the distributed test architecture introduces inherent limitations into testing and these limitations cannot be avoided unless some mechanism can be established to synchronise the local testers [18]. As a result, if it is not possible to synchronise the local testers then any test generation technique that returns test sequences that overcome the limitations imposed by the distributed test architecture must either be incomplete (misses some 'faults') or restricted to a special class of FSMs. Naturally, similar observations apply to IOTSs. This led to the definition of a new implementation relation for FSMs [18]; in this section we focus on the corresponding implementation relations defined for IOTSs. We need to include information about ports into models.

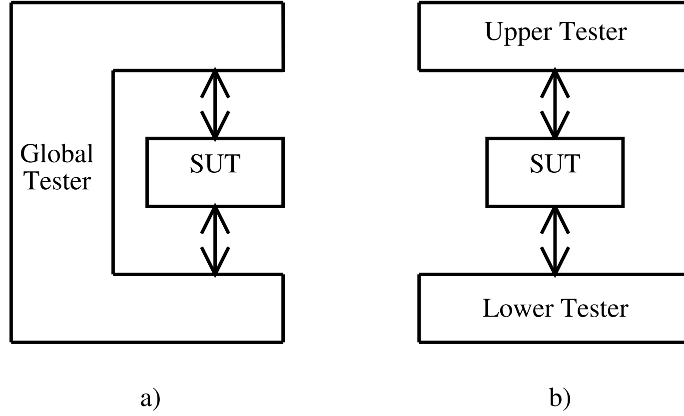


**Fig. 4.** Testing architectures in systems with multiple ports

**Definition 5.** *A distributed $\mathcal{IOTS}$ (d$\mathcal{IOTS}$) is a pair $(M, \mathcal{P})$, where $M = (Q, I, O, T, q_{in})$ is an $\mathcal{IOTS}$ and $\mathcal{P}$ is the set of ports. We partition $I$ into pair-wise disjoint sets $I_p$, for all $p \in \mathcal{P}$, containing those inputs that can be received at port $p$. Similarly, $O$ is partitioned into pair-wise disjoint sets $O_p$, for all $p \in \mathcal{P}$, containing those outputs that can be produced at port $p$.*

*$\mathcal{A}ct_p$ denotes the set of observations that can be made at $p$, that is, $\mathcal{A}ct_p = I_p \cup O_p \cup \{\delta\}$.*

*Example 3.* Let us consider the $\mathcal{IOTS}$ depicted in Figure 3, which is actually a $d\mathcal{IOTS}$. The system has three different ports that correspond to the laboratory, the image diagnosis section and the consultations. These ports are connected to the central server where information related to patients is stored. The different types of lines used to draw the transitions are related to the different ports: solid for the doctor's office, dashed for the image diagnosis office and dotted for the laboratory office.

Given port $p$ and a (global) trace $\sigma \in \mathcal{A}ct^*$, we let $\pi_p(\sigma)$ denote the projection of $\sigma$ onto port $p$ and this is called a *local trace*.

**Definition 6 (Projection onto port $p$).** *Let $p \in \mathcal{P}$ and $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. We let $\pi_p(\sigma)$ denote the projection of $\sigma$ onto port $p$ and $\pi_p(\sigma)$ is called a* local trace*. Formally,*

$$\pi_p(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ a\pi_p(\sigma') & \text{if } \sigma = a\sigma' \wedge a \in \mathcal{A}ct_p \\ \pi_p(\sigma') & \text{if } \sigma = a\sigma' \wedge a \in \mathcal{A}ct \setminus \mathcal{A}ct_p \end{cases}$$

*Given $\sigma, \sigma' \in \mathcal{A}ct^*$ we write $\sigma \sim \sigma'$ if $\sigma$ and $\sigma'$ cannot be distinguished when making local observations, that is, for all $p \in \mathcal{P}$ we have that $\pi_p(\sigma) = \pi_p(\sigma')$.*

Note that quiescence is observed at all ports.

## 4 Implementation Relations

Recall that in the distributed test architecture, there is a separate local tester at each port. These testers make local observations and the local observations are used in order to produce a test verdict such as pass (if the observed behaviour is consistent with the specification) or fail (if the observed behaviour is not consistent with the specification). The initial focus was on two main alternatives. In the first of these alternatives, the local tester at port $p$ produces a local verdict $v_p$: the tester determines whether the local observation at $p$ is one allowed by the specification. The local verdicts are then combined, with the overall verdict being fail if and only if one or more of the local verdicts are fail. This leads to the following implementation relation [17].

**Definition 7 (The pdioco implementation relation).** *Let $i, s$ be $d\mathcal{IOTS}$s with port set $\mathcal{P}$. We write $i$* **pdioco** *$s$ if for every trace $\sigma \in \mathcal{T}r^*(i)$ and for every port $p \in \mathcal{P}$ there exists some trace $\sigma' \in \mathcal{T}r^*(s)$ such that $\pi_p(\sigma) = \pi_p(\sigma')$.*

Let us suppose that $\sigma$ is a global trace. Clearly, $\sigma$ uniquely defines the corresponding local traces but, in addition, the converse is not the case: there may be some different global trace $\sigma'$ that has the same set of local traces. It is therefore unsurprising that **pdioco** is strictly weaker than **ioco**.

**Proposition 1.** *Let $i, s$ be $d\mathcal{IOTS}$s. We have that $i$* **ioco** *$s$ implies $i$* **pdioco** *$s$. However, there exist processes $s$ and $i$ such that $i$* **pdioco** *$s$ but where we do not have that $i$* **ioco** *$s$.*

A practical benefit of **pdioco** is that the test infrastructure for such an implementation relation may be relatively simple: each local tester records its local verdict and these local verdicts are either sent to a central tester that combines them or are locally stored and combined later. Thus, the complexity of the *oracle problem*[4] is essentially the same as that of **ioco** (there is a multiplier of $|\mathcal{P}|$). Note also that it has been shown that $i$ **pdioco** $s$ holds if and only if, for every $p \in \mathcal{P}$, the projection of $i$ onto $p$ conforms, under **ioco**, to the projection of $s$ onto $p$ [17].

Although **pdioco** is appealing, the local testers might have observed projections of different global traces of the specification. As a result, the verdict might be pass despite the global trace that occurred being very different from any global trace of the specification. For example, consider the global trace $?app_doctor$ $!date_doctor?test_presc?app_smear_test?app_mammography\,!date_mammography$ $!date_smear_test$. This global trace is not allowed by our specification. However, the projection of this trace onto each port ($?app_doctor!date_doctor?test_presc$, $?app_smear_test!date_smear_test$ and $?app_mammography!date_mammography$) will lead to a pass verdict. Testing can be strengthened by allowing the local testers to log their observations (local traces), with these logs being brought together after testing is complete. This leads to a different implementation relation [16,17].

[4] The oracle problem is the problem of deciding whether an observation made in testing is one allowed by the specification.

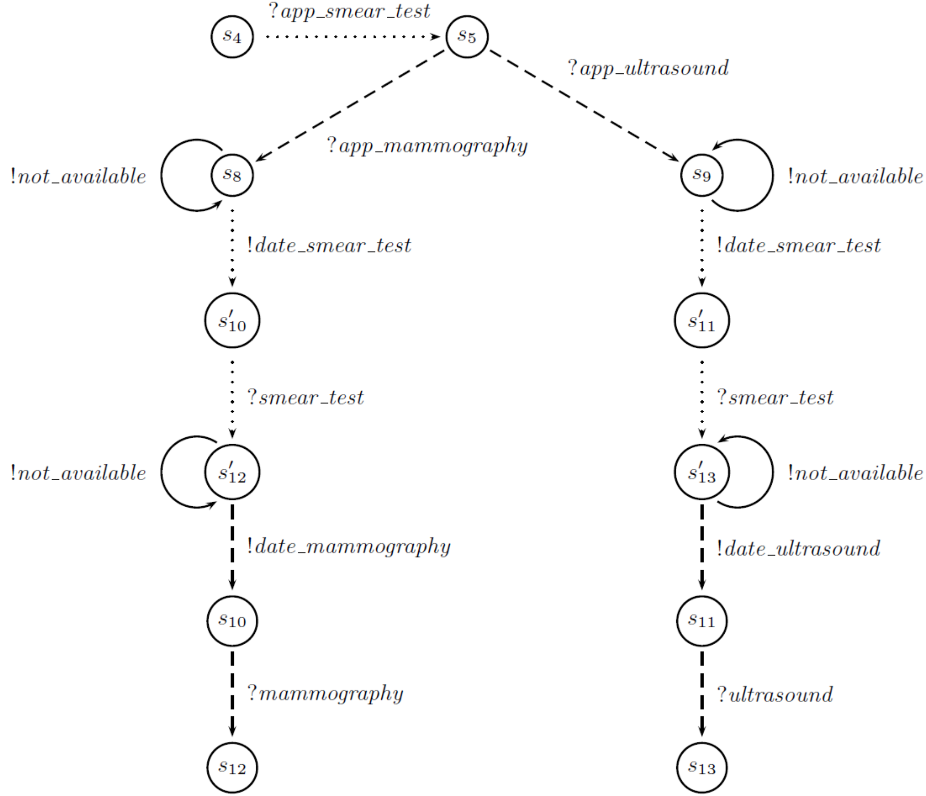


**Fig. 5.** A variant of the protocol

**Definition 8 (The dioco implementation relation).** *Let $i, s$ be d$\mathcal{IOTS}$s. We write $i$* **dioco** *$s$ if for every trace $\sigma$ such that $i \stackrel{\sigma}{\Longrightarrow} i'$ for some $i'$ that is in a stable state, there exists a trace $\sigma'$ such that $s \stackrel{\sigma'}{\Longrightarrow} s'$ and $\sigma' \sim \sigma$.*

*Example 4.* Let us consider the specification presented in Figure 3. If we replace its subgraph starting at $s_4$ and ending at $s_{12}$ and $s_{13}$ by the subgraph depicted in Figure 5 we obtain an alternative protocol. This new protocol does not conform to the original one with respect to **dioco**. For example, if we consider the trace reaching the stable state $s'_{10}$ in this new protocol, $\sigma = ?app_doctor!date_doctor$ $?test_presc?app_smear_test?app_mammography!date_smear_test$, there does not exist any trace $\sigma'$ in the original model such that $\sigma' \sim \sigma$. This is due to the fact that, in the original model, the projection of the traces corresponding to the laboratory office and reaching the transitions labelled with $?app_mammography$ present the input $?smear_test$. This action does not appear in $\sigma$. However, the changes included in the new protocol do not modify the original order of the

actions at the different ports and, therefore, it does conform to the original one if we use **pdioco**.

**Proposition 2.** *There exist $d\mathcal{IOTS}$s $s$ and $i$ such that $i$* **dioco** *$s$ but not $i$* **ioco** *$s$. There also exist $d\mathcal{IOTS}$s $s$ and $i$ such that $i$* **pdioco** *$s$ but not $i$* **dioco** *$s$.*

Although **dioco** has the advantage of being stronger than **pdioco**, it has the disadvantage that one can no longer express the oracle problem in terms of separate instances of the oracle problem for the local testers. In fact, even for a deterministic FSM specification, the oracle problem becomes NP-Complete [10].

Notice that the definition of **dioco** only considers traces that reach stable states of the SUT. The reason for this is that the local testers can effectively 'stop testing' at such stable states: the local testers all observe quiescence at the end of the trace. In practice, the local testers can keep on observing outputs until a stable state is reached and then determine that the state was stable through a sufficiently long timeout.

This approach, of only considering traces that reach stable states, has the benefit of relatively simplicity and leads to an implementation relation that is defined in a similar way to **ioco**. However, **dioco** can be unsuitable if a process is output divergent. To see why this is the case, consider some trace $\sigma$ of the SUT that reaches a quiescent state and an infinite extension $\sigma.\sigma'$ such that none of the states after $\sigma$ are stable: for every non-empty prefix $\sigma''$ of $\sigma'$ we have that $\sigma.\sigma''$ does not reach a stable state. The above definition of **dioco** does not consider any of these $\sigma.\sigma''$, even if they are clearly 'different' from the traces of the specification.

An alternative approach has been defined in terms of *observations* of a process; these correspond to tuples of local traces that might be observed when interacting with the process. Essentially, when a global trace occurs, each local tester observes a prefix of the corresponding local trace (it observes the entire local trace if it waits long enough). In the following, given a (local) trace $\sigma_p$ we let $pref(\sigma_p)$ denote the set of prefixes of $\sigma_p$.

**Definition 9 (Observation).** *Given $d\mathcal{IOTS}$ $r$ with $m$ ports, we say that $obs = (\sigma_1, \ldots, \sigma_m)$ is an* observation of *$r$ if there exists a global trace $\sigma \in \mathcal{T}r^*(r)$ such that for all $p \in \mathcal{P}$, we have that $\sigma_p \in pref(\pi_p(\sigma))$. We let $Obs(r)$ denote the set of possible observations of $r$.*

*Given IOTS $r'$ we say that obs is* allowed by *$r'$ if and only if $obs \in Obs(r')$.*

If one considers the above definition and a specification $s$, then we can give an observation *obs* verdict *pass* if and only if *obs* is allowed by $s$. The idea simply is that although the testers do not know that the local traces they have observed are all projections of the same global trace of the SUT, they do know that they are all prefixes of projections of a global trace of the SUT.

We can now define an alternative implementation relation on the basis of the above: it essentially says that an SUT conforms to a specification if and only if all observations regarding the SUT are also observations regarding the specification [15].

**Definition 10 (The $\mathbf{dioco}_o$ implementation relation).** *Given $d\mathcal{IOTS}$s $i$ and $s$ with the same input and output alphabets and the same set of ports, we write $i$ $\mathbf{dioco}_o$ $s$ if and only if $Obs(i) \subseteq Obs(s)$.*

Note that the oracle problem for $\mathbf{dioco}_o$ is also NP-Complete [15]. The above implementation relation is suitable for processes that are output-divergent and is equivalent to **dioco** if the processes are not output-divergent [15].

**Proposition 3.** *Given $d\mathcal{IOTS}$s $i$ and $s$ that are not output-divergent, $i$ **dioco** $s$ if and only if $i$ $\mathbf{dioco}_o$ $s$.*

The implementation relation $\mathbf{dioco}_o$ is thus a conservative generalisation of **dioco**. A different conservative generalisation of **dioco** has been defined in terms of infinite traces of processes [17]. However, this alternative generalisation has been shown to be too strong in the sense that an implementation $i$ might fail to be a correct implementation of a specification $s$ even though no finite observation can distinguish the SUT and specification [15].

## 5 Related and Future Work

The focus of this paper has been on defining suitable implementation relations, which formalise what it means for an SUT to be a correct implementation of a $d\mathcal{IOTS}$. Such implementation relations can support systematic testing but they do not, on their own, address the problem of generating test cases for use in testing. There have been two main approaches to test generation for testing in the Distributed Test Architecture. One class of approaches, developed for testing from an FSM, involves producing test sequences that have no controllability and/or observability problems (see, for example, [3,23]). Naturally, these techniques lack generality (there are FSMs for which there is no such test sequence) but are potentially powerful where they can be applied. A second class of approaches allows the local testers to exchange synchronisation messages and typically aims to minimise the number of messages or communications channels required in order to overcome controllability and/or observability problems in a given test sequence (see, for example, [22,33]).

Some work has taken into account the nature of distributed testing during test generation. One proposal is to generate test cases in the form of tuples of (local) test cases: one local test case per port [17]. It is then possible to check whether such a test case introduces controllability and/or observability problems. It is unclear, however, how one might generate suitable test cases that are guaranteed to be free from such problems; it may be best to simply generate test cases and accept that controllability problems may lead to non-determinism in the interaction between a test case and an SUT even if the SUT is deterministic. A second disadvantage of this approach, in which one generates a separate local test case for each port, is that it is more difficult to relate these test cases to test objectives, such as covering part of a model. If one is interested in generating test cases to cover part of a model then one might instead represent

test generation as a multi-player game problem, although it transpires that the existence of test cases guaranteed to lead to, for example, a given state being reached is undecidable [8]. A third approach limits the aim of testing to finding faults that can be found using controllable test cases [14] and returns test suites that find all such faults (subject to the standard FSM testing assumption that we have a known upper bound on the number of states of the SUT).

Recent work by Huang and Peleska [20] has devised a model-independent approach to testing. They observe that the semantics of a state-based model is a set $L$ of traces and if the original model is finite-state then $L$ is regular. The semantics $L$ thus induces an LTS $LTS(L)$, which can be defined largely through the use of Nerode-equivalence (two traces $\sigma$ and $\sigma'$ reach the same state of the induced LTS if they have the same set of continuations in $L$). Testing can then be based on the induced model $LTS(L)$. This approach addresses a weakness of test generation based on coverage, which is that two models may be equivalent (have the same semantics) and yet lead to different test suites. It also moves coverage away from the coverage of syntax and towards the coverage of semantics. It would be interesting to adapt this approach to distributed testing and there appear to be at least two possible routes. First, one could define a language whose elements are tuples of (local traces) and define a notion similar to Nerode-equivalence for such a language. Alternatively, one could extend the language $L$ defined by an LTS by including all traces that are observationally equivalent to traces of $L$ and use this extended language as the basis for inducing an LTS.

## 6 Conclusions

Although testing is an important part of software development, it is often manual and so expensive and error-prone. If there is a model (specification) of the required behaviour of the SUT and this model has a formal semantics then there is potential to base systematic test generation on this model. However, it is important to use a suitable implementation relation since otherwise, for example, testing might incorrectly suggest that a correct SUT is faulty or a faulty SUT is correct.

Most approaches to model-based testing (MBT) assume that there is a single tester that interacts with the SUT and can observe the global trace produced by the SUT. Sometimes, however, the SUT has multiple physically distributed ports and there is a local tester at each port. If the distributed test architecture is used then no tester can observe the global trace produced by the SUT and verdicts must instead be based on local traces (projections of the global trace).

We have described several different implementation relations defined for the distributed test architecture. The simplest approach is for each local tester to compare its observation (local trace) against the local traces allowed by the specification, with the overall verdict being fail if and only if one of these local verdicts is fail. The corresponding implementation relation **pdioco** is equivalent to the one produced if one compares the projections of the SUT and specifica-

tion using the standard implementation relation **ioco**. However, we have seen that **pdioco** can hold between an SUT and a specification even if the SUT has behaviours (global traces) that are very different from those of the specification. This motivated the definition of a stronger implementation relation, **dioco**, that corresponds to a scenario in which the local tester observe local traces and the local traces are brought together. We have also seen that this can be generalised to remove the constraint that processes are not output-divergent.

The implementation relations provide a formal basis for testing within the distributed test architecture. However, much remains to be done. For example, we have also seen that there has been relatively little work on test generation algorithms that target these implementation relations. In this context, it may be possible to extend the approach of Huang and Peleska [20], which bases test generation on a model induced by the language defined by the specification.

## References


1. C. Braunstein, A. E. Haxthausen, W.-L. Huang, F. Hübner, J. Peleska, U. Schulze, and L. V. Hong. Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In *16th Int. Conf. on Formal Engineering Methods, ICFEM'14, LNCS 8829*, pages 380–395. Springer, 2014.
2. L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *Information and Software Technology*, 41(11–12):767–780, 1999.
3. W. Chen and H. Ural. Synchronizable checking sequences based on multiple UIO sequences. *IEEE/ACM Transactions on Networking*, 3:152–157, 1995.
4. R. Dssouli and G. von Bochmann. Error detection with multiple observers. In *5th WG6.1 Int. Conf. on Protocol Specification, Testing and Verification, PSTV'85*, pages 483–494. North-Holland, 1985.
5. R. Dssouli and G. von Bochmann. Conformance testing with multiple observers. In *6th WG6.1 Int. Conf. on Protocol Specification, Testing and Verification, PSTV'86*, pages 217–229. North-Holland, 1986.
6. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ACM SIGSOFT Symposium on Software Testing and Analysis, ISSTA'02*, pages 112–122. ACM Press, 2002.
7. W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011.
8. R. M. Hierons. Reaching and distinguishing states of distributed systems. *SIAM Journal on Computing*, 39(8):3480–3500, 2010.
9. R. M. Hierons. The complexity of asynchronous model based testing. *Theoretical Computer Science*, 451:70–82, 2012.
10. R. M. Hierons. Oracles for distributed testing. *IEEE Transactions on Software Engineering*, 38(3):629–641, 2012.
11. R. M. Hierons. Overcoming controllability problems in distributed testing from an input output transition system. *Distributed Computing*, 25(1):63–81, 2012.
12. R. M. Hierons. Implementation relations for testing through asynchronous channels. *The Computer Journal*, 56(11):1305–1319, 2013.
13. R. M. Hierons. Combining centralised and distributed testing. *ACM Transactions on Software Engineering and Methodology*, 24(1):article 5, 2014.

14. R. M. Hierons. Generating complete controllable test suites for distributed testing. *IEEE Transactions on Software Engineering*, 41(3):279–293, 2015.
15. R. M. Hierons. A more precise implementation relation for distributed testing. *Computer Journal*, 59(1):33–46, 2016.
16. R. M. Hierons, M. G. Merayo, and M. Núñez. Implementation relations for the distributed test architecture. In *Joint 20th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'08, and 8th Int. Workshop on Formal Approaches to Software Testing, FATES'08, LNCS 5047*, pages 200–215. Springer, 2008.
17. R. M. Hierons, M. G. Merayo, and M. Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35–62, 2012.
18. R. M. Hierons and H. Ural. The effect of the distributed test architecture on the power of testing. *The Computer Journal*, 51(4):497–510, 2008.
19. H.-M. Hörcher and J. Peleska. Using formal specifications to support software testing. *The Software Quality Journal*, 4(4):309–327, 1995.
20. W. Huang and J. Peleska. Model-based testing strategies and their (in)dependence on syntactic model representations. *International Journal on Software Tools for Technology Transfer*, 20(4):441–465, 2018.
21. Joint Technical Committee ISO/IEC JTC 1. *International Standard ISO/IEC 9646-1. Information Technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts.* ISO/IEC, 1994.
22. G.-V. Jourdan, H. Ural, and H. Yenigün. Minimizing coordination channels in distributed testing. In *26th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'06, LNCS 4229*, pages 451–466. Springer, 2006.
23. G. Luo, R. Dssouli, and G. von Bochmann. Generating synchronizable test sequences based on finite state machine with distributed ports. In *6th IFIP Workshop on Protocol Test Systems, IWPTS'93*, pages 139–153. North-Holland, 1993.
24. J. Peleska. Industrial-strength model-based testing - state of the art and current challenges. In *8th Workshop on Model-Based Testing, MBT'13, EPTCS 111*, pages 3–28, 2013.
25. J. Peleska. Model-based avionic systems testing for the airbus family. In *23rd IEEE European Test Symposium, ETS'18*, pages 1–10. IEEE Computer Society, 2018.
26. J. Peleska, A. Honisch, F. Lapschies, H. Löding, H. Schmid, P. Smuda, E. Vorobev, and C. Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In *23rd Int. Conf. on Testing Software and Systems, ICTSS'11, LNCS 7019*, pages 146–161. Springer, 2011.
27. J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.
28. R. Sachtleben and J. Peleska. Effective grey-box testing with partial FSM models. *Software Testing, Verification and Reliability*, 32(2), 2022.
29. B. Sarikaya and G. von Bochmann. Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications*, 32:389–395, 1984.
30. J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.
31. H. Ural and D. Whittier. Distributed testing without encountering controllability and observability problems. *Information Processing Letters*, 88(3):133–141, 2003.

32. M. Weiglhofer and F. Wotawa. Asynchronous input-output conformance testing. In *33rd Annual IEEE Computer Software and Applications Conference, COMPSAC'09*, pages 154–159. IEEE Computer Society, 2009.
33. W.-J. Wu, W.-H. Chen, and C.Y. Tang. Synchronizable test sequence for multi-party protocol conformance testing. *Computer Communications*, 21(13):1177–1183, 1998.