Distributed Computing manuscript No.
(will be inserted by the editor)

Implementation relations and test generation for systems with distributed interfaces

Robert M. Hierons · Mercedes G. Merayo · Manuel Núñez

Received: date / Accepted: date

Abstract Some systems interact with their environment at physically distributed interfaces called ports and we separately observe sequences of inputs and outputs at each port. As a result we cannot reconstruct the global sequence that occurred and this reduces our ability to distinguish different systems in testing or in use. In this paper we explore notions of conformance for an input output transition system that has multiple ports, adapting the widely used **ioco** implementation relation to this situation. We consider two different scenarios. In the first scenario the agents at the different ports are entirely independent. Alternatively, it may be feasible for some external agent to receive information from more than one of the agents at the ports of the system, these local behaviours potentially being brought together and here we require a stronger implementation relation. We define implementation relations for these scenarios and prove that in the case of a single-port system the new implementation relations are equivalent to **ioco**. In addition, we define what it means for a test case to be controllable and give an algorithm that decides whether this condition holds. We give a test generation algorithm to produce sound and complete test suites. Finally, we study two implementation relations to deal with partially specified systems.

Keywords Formal approaches to testing; Systems with distributed ports; Formal methodologies to develop distributed software systems.

Research partially supported by the Spanish MICINN project TESIS (TIN2009-14312-C02-01), the UK EPSRC project Testing of Probabilistic and Stochastic Systems (EP/G032572/1), and the Santander-UCM Programme to fund research groups (GR35/10-A - group number 910606).

Robert M. Hierons
Department of Information Systems and Computing,
Brunel University,
Uxbridge, Middlesex, UB8 3PH United Kingdom
E-mail: rob.hierons@brunel.ac.uk

Mercedes G. Merayo and Manuel Núñez
Departamento de Sistemas Informáticos y Computación,
Facultad de Informática,
Universidad Complutense de Madrid, 28040 Madrid, Spain
E-mail: mgmerayo@fdi.ucm.es,mn@sip.ucm.es

1 Introduction

Software systems have reached a high complexity since they have multiple components that are usually built by different developers. As a result, none of the development team members knows the implementation so thoroughly to state, without any additional consideration, that the system is correct. Therefore, it is necessary to apply systematic techniques in order to assess the system correctness. One of the software engineering methodologies to perform this assessment consists of working with an abstract model (specification) showing the desirable behaviour of the system. Then, the correctness of the system is defined in terms of its comparison with the specification: We say that the system *conforms* to the specification if it shows a *similar* behaviour to that of the model. In order to check the conformance of the developed system with respect to the specification, we may use formal testing techniques to extract tests from the specification, each test representing a desirable behaviour that the system must fulfill.

Many systems have a persistent internal state structure and such systems are typically specified or modeled using state based languages such as finite state machines (FSMs) and input output transition systems (`IOTSs`). In addition, state based models are usually used in model-based testing. The area of testing against

a state based model has thus received much attention (see, for example, [39,51,43,8,15–17,2,44,45,48,23,55, 22,47]). The main advantage of using a formal testing approach is that most processes related to testing can be automated (see, for example, [54], for a discussion regarding the advantages of formal testing with respect to *manual* testing).

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 during testing, and there is no global clock then we are testing in the *distributed test architecture* [35]. 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, [49,13,14, 5,40,50,46,53,28]). 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`, although there are costs associated with deploying such a network and it is not always possible to run test cases that have timing constraints [38].

Previous work has largely concentrated on trying to overcome the limitations imposed on testing by the use of the distributed test architecture. However, if the SUT is to be used in a context in which there are separate agents at its ports and these agents do not directly communicate with one another then the requirements placed on the SUT do not correspond to traditional implementation relations. Thus, even if we are not testing in the distributed test architecture we should recognise the effect of there being distributed interfaces. In particular, if we are not using the distributed test architecture and we test the SUT using a method based on a standard implementation relation, testing may return an incorrect verdict. For example, this is the case if we use **ioco** [51], a well-established implementation relation for the non-distributed architecture where a SUT is correct with respect to a specification if for every sequence of actions that both the SUT and the specification can produce, we have that the outputs that the SUT can show after performing the sequence are a subset of those that the specification can show. In addition, a refinement process that does not reflect the weaker requirements that result from there being distributed interfaces may be suboptimal. It is therefore important to devise new implementation relations for this situation.

Work on testing in the distributed test architecture has mainly focussed on testing from DFSMs and two effects have been identified. First, *controllability* problems may occur, where a tester cannot know when to apply an input. Let us suppose, for example, that a test case starts with input $?i_p$ at port p, this should lead to output $!o_p$ at p only and this is to be followed by input $?i_q$ at $q \neq p$. The tester at q cannot know when to send $?i_q$ since it does not observe either the input or output at p. This is illustrated in Figure 1 in which each vertical line represents a timeline, time progressing as we move down a line.

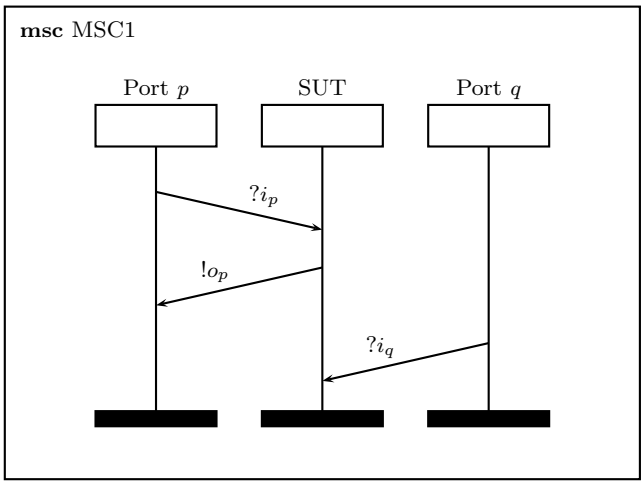


Fig. 1 A controllability problem.

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 $?i_p$ at p, this is expected to lead to output $!o_p$ at p only, this is to be followed by input $?i_p$ at p and this should lead to output $!o_p$ at p and $!o_q$ at $q \neq p$. The tester at p expects to observe $?i_p!o_p?i_p!o_p$ and the tester at q expects to observe $!o_q$. This is still the case if the SUT produces $!o_p$ and $!o_q$ in response to the first input and $!o_p$ in response to the second input: Two faults have masked one another in the sequence used but could lead to failures in different sequences. The two indistinguishable scenarios are illustrated in Figures 2 and 3. The essential problem is that we cannot reconstruct the input/output sequence that occurred from the local projections: It could have been any interleaving of these. Interestingly, this situation relates to the case when testing a system with a buffer since all we know in this case is that the outputs were produced before they were observed and the inputs were received by the SUT after they were sent [32,33]. However, the situation is different since the case when testing with a buffer places restrictions on the interleavings that could have occurred due to the asymmetry between input and output. Work on testing in the distributed test architecture has largely concerned finding test sequences without controllability or observability problems (see, for example, [49,13,

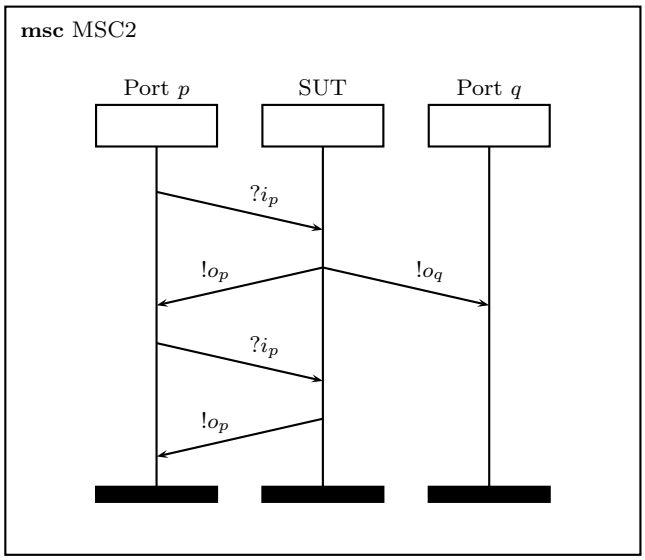


Fig. 2 A global trace involved in an observability problem.

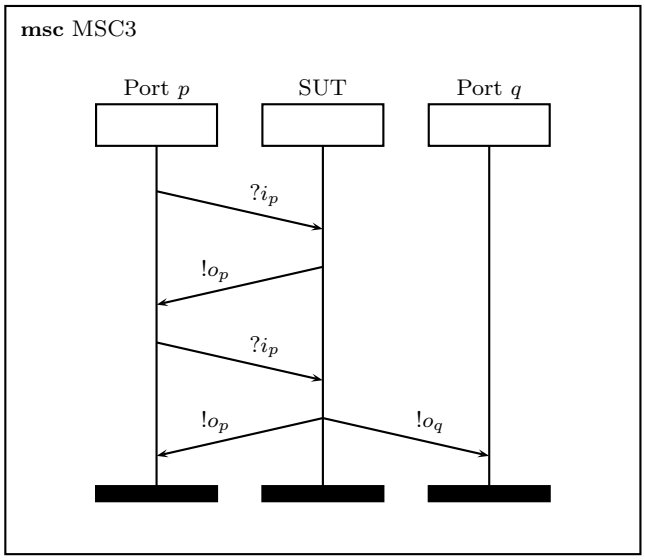


Fig. 3 An indistinguishable global trace.

14,5,40,50,53]) but recent work has characterised the effect of the distributed test architecture on the ability of testing to distinguish between a DFSM specification and a DFSM implementation [29]. This can be seen as defining an implementation relation for DFSMs.

While DFSMs are appropriate for modelling and specifying several important classes of systems, they are less expressive than input output transition systems (IOTSs). For example, an IOTS can be nondeterministic, which is an important feature for distributed systems. In addition, in a DFSM input and output alternate and the state space is finite, while an IOTS need not have these restrictions. This paper investigates implementation relations and testing for IOTSs with multiple ports.

We consider two scenarios. In the first scenario there is an agent at each port of the system and these agents are independent in that no external agent or system can receive information from more than one of them. In order to characterise this scenario, it is sufficient that the local behaviour observed by an agent is consistent with some global behaviour in the specification. We define a new implementation relation, that we call **p-dioco**, based on this idea. In the second scenario there is the possibility that information from two or more of the agents could be received by some external agent/system. As a result the local behaviours observed at the separate ports could be brought together. Consequently, we define a stronger implementation relation, which we call **dioco**. This implementation relation is a generalisation of the implementation relation defined in [24,25] and is strictly stronger than this previous implementation relation. In this paper we prove that **ioco** and **dioco** coincide for single-port systems but that **dioco** is weaker than **ioco** if there are multiple ports. This result, even though expected, shows that our notions are appropriate conservative extensions to the distributed architecture of the classical notions. While **ioco** has been adapted in a number of ways to cope with issues such as time and probability, as far as we know, our previous work [24,25] was the first to define an implementation relation based on **ioco** for testing from an IOTS in the distributed test architecture. The most similar work to ours is the implementation relation **mioco** [7]. This relation has been defined for testing from an IOTS with multiple ports, but it is restricted to the case when there is a single tester that controls and observes all of the ports. This single agent can observe the global behaviour of the system. Interestingly, the concept of the agent at a port only making local observations has been explored in the context of refinement in CSP [36].

Another contribution of this paper is to explore the concept of a *local test case*. In our terminology, a global test case is a single description that contains information about the inputs that will be applied to the SUT and about the outputs that the SUT is expected to produce. In contrast, a local test case is a set of test cases, one for each port. We say that each of these test cases is a *local tester*. Formally, $(t_1, \ldots, t_m)$ is a local test case, in which t_i is a local tester at port i. In this paper we provide a notion of a global test case being controllable. As a starting point we consider [24], where this notion was only defined for a restrictive class of IOTSs and the case where there are only two ports. We show that there are deterministic global test cases that are not controllable and show how a controllable global test case t can be mapped to a controllable local test case

that implements t. We also provide a polynomial time algorithm that determines whether a test case is controllable. If the interactions between the SUT and its environment correspond to controllable test cases then we obtain an alternative implementation relation that we call **c-dioco**. We also define a test generation algorithm to provide alternative characterisations of the implementation relations introduced in this paper.

The previously described framework has a restriction inherited from our previous work [24,25]: All the systems must be *input-enabled*, that is, each state of the system has to provide a certain reaction to each possible input. This restriction simplifies some definitions but it is not essential to the framework. In fact, most work on formal testing considers that SUTs are input-enabled but specifications do not need to be. Therefore, we have studied the situation where we remove the restriction that specifications must be input-enabled. Somewhat surprisingly, there does not exist a unique *natural* implementation relation. We present two implementation relations, discuss their (dis-)advantages, and provide alternative characterisations based on the traces that the processes can perform.

In summary, this paper extends and enhances [24, 25] in the following ways. First, we look at two different testing scenarios and define three implementation relations, two of which are generalisations of those defined in [24,25]. We show how the three implementation relations are related to one another and to **ioco**. Our original notion, that we call **dioco**$_\delta$ in this paper, is strictly weaker than our new implementation relation **dioco**. It also does not have some of the desirable properties, such as being stronger than **p-dioco** and being equivalent to **ioco** when testing single-port systems. We give a polynomial time algorithm that determines whether a test case is controllable. While [24] presents an algorithm for deciding whether a test case is controllable, the work only considered the case where there are two ports and had the restriction that an action in an `IOTS` cannot send output to more than one port. Relaxing the second assumption requires an entirely different algorithm. We define a test generation algorithm that, due to its *soundness* and *completeness*, can be used to provide alternative characterisations of the introduced implementation relations. Our previous work [24] gave a test generation algorithm for one of the implementation relations, for a restricted form of `IOTS`, and in the case where there are only two ports. The two additional implementation relations dealing with non input-enabled specifications are completely new and their definitions are far from trivial. Finally, we consider the general case in which there are $m > 1$ ports while [24,25] only investigated the situation in which there are two ports. This extension is technically straightforward but improves the applicability of the framework. The table given in Figure 4 summarizes the implementation relations appearing in this paper.

The rest of the paper is structured as follows. Section 2 provides preliminary material and in Section 3 we define an implementation relation to be used when the agents at the ports are entirely independent and their observations cannot later be brought together. Section 4 then considers the case in which the observations at separate ports can be brought together later, leading to a stronger implementation relation. Section 5 defines local test cases to be used when there are distributed interfaces and considers the problem of assigning a verdict to a test run. Section 6 firstly describes what it means for a test case to be deterministic and shows that it is not enough to require determinism in test cases to achieve controllability. In addition, this section presents an algorithm that determines whether a test case is controllable. Finally, it also gives an implementation relation for the case when the use of the SUT corresponds to controllable test cases and shows how the three implementation relations are related to one another and to **ioco**. Section 7 then gives a test case generation algorithm to produce sound and complete test suites with respect to the implementation relations defined in this paper. Section 8 presents two implementation relations to deal with non input-enabled specifications. Section 9 reviews some related work in the area of testing distributed systems. Finally, conclusions are drawn in Section 10. We have included an appendix where the interested reader can check the proofs of some of the results appearing in the paper.

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. Then, we review the main differences between *classical* testing and testing in the distributed architecture. Throughout this paper, given a sequence σ we let $pre(\sigma)$ denote the set of prefixes of σ, including the empty sequence ϵ. If A is a set then A^* and A^ω denote the sets of finite and infinite sequences of elements of A, respectively. Let $\sigma \in A^*$ and $a \in A$. We have that σa denotes the concatenation of the sequence σ followed by a

Name	Restrictions on specifications	Location	Description
ioco	none	Definition 4	Standard implementation relation for single-port systems.
p-dioco	input-enabled	Definition 7	Implementation relation for multi-port systems assuming that external agents do not communicate.
dioco$_\delta$	input-enabled	Definition 9	Implementation relation for multi-port systems that considers only finite and quiescent traces. Superseded by **dioco**.
dioco	input-enabled	Definition 11	Implementation relation for multi-port systems that extends **dioco**$_\delta$ to consider infinite traces. Main implementation relation of the paper.
c-dioco	input-enabled	Definition 20	Restriction of **dioco** to controllable tests.
dioco$_1$	none	Definition 21	Implementation relation for multi-port systems where it is assumed that unspecified sequences can lead to any output.
dioco$_2$	none	Definition 22	Implementation relation for multi-port systems where it is assumed that a sequence is specified if there exists an equivalent, up to $\sim$, sequence.

Fig. 4 Implementation relations used in this paper.

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* (IOTS) is defined by a tuple $s = (Q, I, O, T, q_{in})$ 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 an internal (unobservable) action, 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 say that a state $q \in Q$ is *quiescent* if from q it is not possible to produce output without first receiving input. We say that 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 a system s is *divergent* if it can reach a state from which there is an infinite path that contains only internal actions.

As usual, we assume that implementations and specifications are input-enabled.[1] In addition we also assume that all the processes considered in this paper, either implementations or specifications, are not divergent.

An IOTS can be represented by a diagram in which nodes represent states of the IOTS and transitions are represented by arcs between the nodes. In order to distinguish between input and output we usually precede the name of an input by ? and precede the name of an output by !.

[1] If an input cannot be applied in some state of the SUT, then we can assume that there is a response to the input that reports that this input is blocked. We remove this restriction for specifications in Section 8.

We build our results from the theory of **ioco**, which assumes that it is known whether states of the SUT reached in testing are quiescent. Following this assumption, we extend the transition relation with self-loop transitions labeled δ to denote quiescent states.

Definition 2 Let (Q, I, O, T, q_{in}) be an IOTS. We 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, $\mathcal{A}ct = I \cup O \cup \{\delta\}$. Therefore, the set $\mathcal{A}ct^*$ contains all the finite sequences that can be formed by using elements in $\mathcal{A}ct$ while $\mathcal{A}ct^\omega$ denotes the set of infinite sequences that can be formed by using elements in $\mathcal{A}ct$. Every element of $\mathcal{A}ct^* \cup \mathcal{A}ct^\omega$ is called a *trace*.

Traces are often called *suspension traces*, as they can include quiescence, but since this is the only type of trace we consider we simply call them traces. Let us remark that traces cannot contain τ. As usual, we assume that systems cannot block inputs and that the environment cannot block output produced by systems. The following is standard notation in the context of **ioco** (see, for example, [51]).

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 $q_0, \ldots, q_k \in Q$ and $i \geq 0$ such that $q = q_0$, $q' = q_k$, and we have the following transitions $q_0 \xrightarrow{\tau} q_1, \ldots, q_{i-1} \xrightarrow{\tau} q_i$, $q_i \xrightarrow{a} q_{i+1}$, $q_{i+1} \xrightarrow{\tau} q_{i+2}, \ldots, q_{k-1}, \xrightarrow{\tau} q_k$.
3. We write $q \stackrel{\epsilon}{\Longrightarrow} q'$ if there exist $q_0, \ldots, q_k \in Q$, for $k \geq 0$, such that $q = q_0$, $q' = q_k$, and we have the following transitions $q_0 \xrightarrow{\tau} q_1, \ldots, q_{k-1} \xrightarrow{\tau} q_k$.

4. Let $\sigma = a_1 \dots a_k \in \mathcal{A}ct^*$ be a finite trace. We write $q \stackrel{\sigma}{\Longrightarrow} q'$ if there exist $q_0, \dots, q_k \in Q$ such that $q = q_0$, $q' = q_k$ and for all $1 \leq i \leq k$ we have that $q_{i-1} \stackrel{a_i}{\Longrightarrow} q_i$.
5. We write $q \stackrel{\sigma}{\Longrightarrow}$ if there exists $q' \in Q$ such that $q \stackrel{\sigma}{\Longrightarrow} q'$.
6. We write $s \stackrel{\sigma}{\Longrightarrow}$ if $q_{in} \stackrel{\sigma}{\Longrightarrow}$ and we say that σ is a *trace* of s. We let $\mathcal{T}r^*(s)$ denote the set of *finite traces* of s. We let $\mathcal{T}r^\omega(s)$ denote the set of *infinite traces* of s, where an infinite sequence $\sigma = a_1, a_2, \dots \in \mathcal{A}ct^\omega$ is in $\mathcal{T}r^\omega(s)$ if and only if there exist $q_1, q_2, \dots \in Q$ such that $q_{in} = q_1$ and for all $i \geq 1$ we have that $q_i \stackrel{a_i}{\Longrightarrow} q_{i+1}$.
7. We let $\mathcal{T}r(s) = \mathcal{T}r^*(s) \cup \mathcal{T}r^\omega(s)$ denote the set of *traces* of s.

Let $q \in Q$ be a state and $\sigma \in \mathcal{A}ct^*$ be a trace. We introduce the following concepts.

1. q **after** $\sigma = \begin{cases} \{r \in Q | q \stackrel{\sigma}{\Longrightarrow} r\} \text{ if } q \stackrel{\sigma}{\Longrightarrow} \\ \emptyset \qquad \text{otherwise} \end{cases}$
2. $\mathbf{out}(q) = \{o \in O \cup \{\delta\} | q \stackrel{o}{\Longrightarrow}\}$

The function **out** can be extended to deal with sets in the expected way, that is, given $Q' \subseteq Q$ we define $\mathbf{out}(Q') = \cup_{q \in Q'} \mathbf{out}(q)$.

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

Let us note that for every state q, $q \stackrel{\epsilon}{\Longrightarrow} q$ holds. Moreover, $q \stackrel{a}{\longrightarrow} q'$ implies $q \stackrel{a}{\Longrightarrow} q'$. In addition, $\epsilon \in \mathcal{T}r(s)$ for every process s.

Processes and states are effectively the same since we can identify a process with its initial state and we can define a process corresponding to a state q of s by making q the initial state. Thus, we use states and process and their notation interchangeably. It is important to note that finite traces are somehow *contained* in the set of infinite traces in the following important way.

Proposition 1 *Let $s = (Q, I, O, T, q_{in})$ be an* `IOTS` *and $\sigma \in \mathcal{T}r^*(s)$ be a finite trace. Then there exists some $\sigma' \in \mathcal{T}r^\omega(s)$ such that $\sigma' = \sigma\sigma''$ for some σ'' that contains no inputs.*

Proof We can extend the sequence $\sigma = \sigma_0$ to sequences $\sigma_1, \sigma_2, \dots$ in the following way. If σ_i reaches a state s' such that $s' \stackrel{a}{\Longrightarrow} s''$ for some output a and state s'' then we let $\sigma_{i+1} = \sigma_i a$. Otherwise, σ_i reaches a state s' from which s cannot produce output without first receiving input and so we can set $\sigma_{i+1} = \sigma_i \delta$.

This result will be implicitly used during the rest of the paper when we enunciate some results only for infinite traces, since the information collected by finite traces is appropriately encoded into infinite ones. Let us note that this result holds event even if s is not input-enabled. Next we present the standard implementation relation for testing from an `IOTS` [51].

Definition 4 Let i, s be `IOTSs`. We write i **ioco** s if for every $\sigma \in \mathcal{T}r^*(s)$ we have that $\mathbf{out}(i$ **after** $\sigma) \subseteq \mathbf{out}(s$ **after** $\sigma)$.

Let us note that the classical definition of **ioco** does not consider infinite traces, but we will use them in our framework.

2.2 Multi-port input output transition systems

Some systems have multiple interfaces, called ports, at which they interact with their environment. In order to properly specify this kind of system we need to adapt our notion of `IOTS` to this setting where multiple ports are available. Throughout this paper, we 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`.

Definition 5 Let $s = (Q, I, O, T, q_{in})$ be an `IOTS` and $\mathcal{P}(s)$ be the port set of s. If $\mathcal{P}(s) = \{1, \dots, m\}$ then the set I is partitioned into sets $I_1, \dots, I_m$ such that for all $p \in \mathcal{P}(s)$, I_p is the set of inputs that can be received by the SUT at port p. Similarly, given port $p \in \mathcal{P}(s)$, O_p denotes the set of outputs that can be produced by the SUT at p. Each element of O is thus in $(O_1 \cup \{-\}) \times \dots \times (O_m \cup \{-\})$ in which $-$ denotes empty output and $(-, \dots, -)$ is not an element of O. We assume that $I_1, \dots I_m, O_1, \dots, O_m$ are pairwise disjoint. Finally, given port $p \in \mathcal{P}(s)$, Act_p denotes the set of observations that can be made at p, that is, $Act_p = I_p \cup O_p \cup \{\delta\}$.

In the following, if the port set can be inferred from the context we will simply write $\mathcal{P}$ instead of $\mathcal{P}(s)$. Moreover, in the rest of the paper, when we relate two `IOTSs` we will assume that they have the same port set. Let us note that if the same types of values can be received or sent by the SUT at different ports then we can label these in order to ensure that the sets are disjoint. In addition, it is straightforward to extend the results to specifications and systems that have different port sets. For the sake of clarity, all examples in this paper use two ports, called U and L for the ports connected to the upper and lower testers, respectively. However, results are enunciated and proved for the general case where there are $m > 1$ ports.

Example 1 In Figure 5 we sketch, as an `IOTS`, the *server side* of a simple protocol to perform majority voting. We

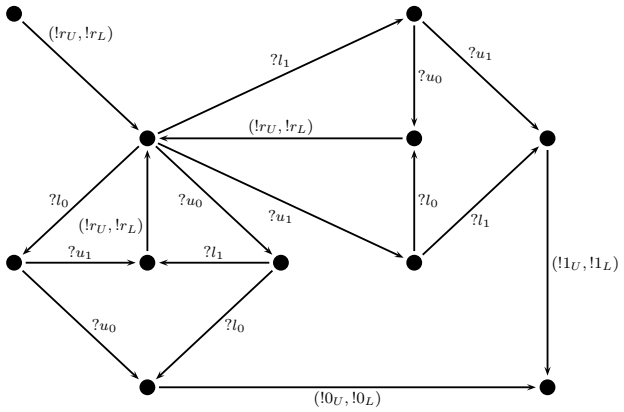


Fig. 5 A simple distributed majority voting protocol.

will use this system as a running example to illustrate some of the concepts that we will introduce in the paper. There are two distributed parties that are asked to vote either 0 or 1 (this is indicated by the transition labelled by $(!r_U, !r_L)$). Then the server receives, as inputs, the answers from each party $?u_x$ and $?l_x$. If they voted the same, then the server sends a message to the parties, as an output, with the result $(!0_U, !0_L)$ or $(!1_U, !1_L)$, and if they voted differently, then the server resends a request for votes, as the output $(!r_U, !r_L)$.

The IOTS defining the system is $s = (Q, I, O, T, q_{in})$, where Q, the states of the system, are given by the different bullets, q_{in}, the initial state, is the state at the top-left, $I = \{?l_0, ?l_1, ?u_0, ?u_1\}$ is the set of inputs, $O = \{(!r_U, !r_L), (!0_U, !0_L), (!1_U, !1_L)\}$ is the set of outputs, and T, the set of transitions, is given by the directed arcs of the graph. For the sake of clarity we have not added *irrelevant* transitions. Specifically, in order to obtain an input-enabled specification we need to add a transition $(q, ?i, q)$ to each state q not having an outgoing transition labeled by the input $?i$.

If we see the previous IOTS as a multi-port IOTS we have two different ports, that is, $\mathcal{P}(s) = \{L, U\}$, so that $I_L = \{?l_0, ?l_1\}$, $I_U = \{?u_0, ?u_1\}$, $O_L = \{!r_L, !0_L, !1_L\}$, and $O_U = \{!r_U, !0_U, !1_U\}$.

Next, we introduce some additional notation.

Definition 6 Let $s = (Q, I, O, T, q_{in})$ be an IOTS with port set $\mathcal{P}(s) = \{1, \ldots, m\}$. Given a port p and an output $y = (!o_1, \ldots, !o_m) \in O$ we let $y]_p$ denote $!o_p$.

Let $p \in \mathcal{P}(s)$ and $\sigma \in Act^* \cup Act^\omega$ be a (finite or infinite) sequence of visible actions. We consider that $\pi_p(\sigma)$ denotes the projection of σ onto port p and $\pi_p(\sigma)$ is called a *local trace*. Formally,

1. $\pi_p(\epsilon) = \epsilon$
2. if $a \in I_p \cup \{\delta\}$ then $\pi_p(a\sigma) = a\pi_p(\sigma)$
3. if $a \in O$ and $a]_p = !o_p \neq -$ then $\pi_p(a\sigma) = !o_p\pi_p(\sigma)$
4. if $a \in I_{p'}$ for $p' \in \mathcal{P}(s)$ with $p' \neq p$ or $a \in O$ with $a]_p = -$, then $\pi_p(a\sigma) = \pi_p(\sigma)$.

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

If an output tuple has output $!o_p$ at one port p only, that is, we have $(-, \ldots, -, !o_p, -, \ldots, -)$, then we often simply represent this as $!o_p$. We will use this notation mainly in the IOTSs that we give to illustrate the differences between the introduced implementation relations.

Example 2 Let us consider our running example (see Example 1). The following are some of the finite traces of the protocol:

$\sigma_1 = (!r_U, !r_L)?l_1?u_1$
$\sigma_2 = (!r_U, !r_L)?u_1?l_1(!1_U, !1_L)$
$\sigma_3 = (!r_U, !r_L)?l_0?u_1(!r_U, !r_L)?u_0?l_1(!r_U, !r_L)?u_0$

Figure 6 shows two infinite traces of the protocol. The corresponding projections on port L are:

$\pi_L(\sigma_1) = !r_L?l_1$
$\pi_L(\sigma_2) = !r_L?l_1!1_L$
$\pi_L(\sigma_3) = !r_L?l_0!r_L?l_1!r_L$
$\pi_L(\sigma_4) = !r_L?l_1!r_L?l_1!r_L?l_1!r_L?l_1 \ldots$
$\pi_L(\sigma_5) = !r_L?l_1!r_L?l_0!r_L?l_1!r_L?l_0 \ldots$

Let us consider now the finite trace

$\sigma_6 = (!r_U, !r_L)?u_1?l_0(!r_U, !r_L)?u_0?l_1(!r_U, !r_L)?u_0$

In this case we have that $\sigma_3 \sim \sigma_6$.

There are two standard architectures [35], shown in Figure 7, for testing a SUT that has multiple interfaces/ports. In the local test architecture a global tester interacts with all of the ports of the SUT. In the distributed test architecture there is a separate local tester at each port and the local tester at port p is responsible for providing input at p and only observes quiescence and the input and output at p.

In order to apply tests, a *global tester* that observes all of the ports observes a trace in $\mathcal{A}ct^*$. When testing in the distributed test architecture, each local tester observes a sequence of actions at its port and so it is not possible to distinguish between two traces if they have the same projections at each port. Consequently, equality is not suitable for comparing traces, and it is necessary to take into account the set of traces induced by the relation $\sim$.

$$\sigma_4 = (!r_U, !r_L)?l_1?u_0(!r_U, !r_L)?l_1?u_0(!r_U, !r_L)?l_1?u_0(!r_U, !r_L)?l_1?u_0 \ldots$$
$$\sigma_5 = (!r_U, !r_L)?l_1?u_0(!r_U, !r_L)?u_1?l_0(!r_U, !r_L)?l_1?u_0(!r_U, !r_L)?u_1?l_0 \ldots$$

Fig. 6 Two infinite traces.

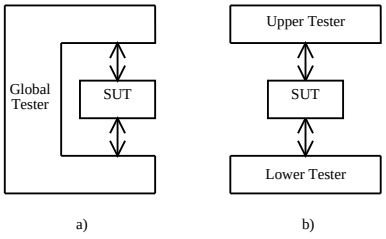


Fig. 7 The local and distributed test architectures

3 Independent agents

In this section we consider the context in which the agents at the ports of the SUT are entirely independent: No external agent or system can receive information regarding observations made at more than one port of the SUT. Thus, an agent at a port can observe only the local trace at that port but has no information about the behaviour of the rest of the SUT. In determining whether the behaviour of the SUT is acceptable, all the agents can do is compare the observed local trace with the local traces that can be produced by the specification. This idea leads to the new implementation relation **p-dioco**.

Definition 7 Let i, s be `IOTSs` with port set $\mathcal{P}$. We write i **p-dioco** s if for every (finite) 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 $\pi_p(\sigma) = \pi_p(\sigma')$.

The definition of **p-dioco** allows the tester at a port p to compare its observation $\pi_p(\sigma)$ with all observations that might be made at p if the SUT behaves like s. For port p the definition therefore quantifies over all finite traces of s. As a result, the trace σ' used in Definition 7 might be different for different ports and this is a natural consequence of the agents being entirely independent; they cannot exchange information. Let us suppose, for example, that s is a process in which the input of $?i_U$ at U is either followed by $(!o_U, !o_L)$ or by $(!o'_U, !o'_L)$. If the sequence $?i_U!o_U$ is observed at U and $!o'_L$ is observed at L, then each agent observes behaviour that is consistent with projections of traces of the specification s and yet the pair of traces is not consistent with any single trace of s. Therefore, these observations are admissible in the context of **p-dioco**. In Section 4 we will give a stronger implementation relation that can be used when we require a trace of the SUT to be observationally equivalent to a single trace of the specification.

We use the following property of **ioco** to reason about the relationship between **ioco** and **p-dioco**. In the proof of the next result we will use the fact that all the systems are input-enabled (without this restriction, the result does not hold).

Proposition 2 *Let i, s be `IOTSs`. We have that i* **ioco** *s if and only if $\mathcal{T}r^*(i) \subseteq \mathcal{T}r^*(s)$.*

Proof Let us assume that i **ioco** s and so we require to prove that if σ is a finite trace of i then σ is a trace of s. We will prove the result by induction on the length of σ.

The base case, $\sigma = \epsilon$, clearly holds. Now, let us assume that the result holds for every trace of length less than k, for some $k > 0$, and let σ be a trace of i with length k. We must have some $a \in \mathcal{A}ct$ and trace σ' such that $\sigma = \sigma' a$. Since σ' has length less than k, by the inductive hypothesis we have that σ' is a trace of s. Now, let us consider the action a. If a is an input then the result follows from the fact that s is input-enabled. If instead a is either an output or δ then the result follows from the fact that $a \in out(i \textbf{ after } \sigma') \subseteq out(s \textbf{ after } \sigma')$. The result therefore follows.

Now, let us assume that $\mathcal{T}r^*(i) \subseteq \mathcal{T}r^*(s)$ and we require to prove that i **ioco** s. Given $\sigma \in \mathcal{T}r^*(s)$ and $a \in \textbf{out}(i \textbf{ after } \sigma)$ we have that $\sigma a \in \mathcal{T}r^*(i)$ and so $\sigma a \in \mathcal{T}r^*(s)$. Thus, $a \in \textbf{out}(s \textbf{ after } \sigma)$ as required.

Naturally, the implementation relation **p-dioco** is strictly weaker than the implementation relation **ioco** as the following result shows.

Proposition 3 *Let i, s be IOTSs. We have that i* **ioco** *s implies i* **p-dioco** *s. However, there exist processes s and i such that i* **p-dioco** *s but where we do not have that i* **ioco** *s.*

Proof The first part follows immediately from Proposition 2. For the second part, let us consider s_1 and i_1 depicted in Figure 8. Clearly, i_1 **p-dioco** s_1 since it is not possible to distinguish between $!o_U!o_L$ and $!o_L!o_U$ when using **p-dioco** and so the result follows from the fact that i_1 **ioco** s_1 does not hold.

The proof of the following result is straightforward.

Proposition 4 *Let i, s be single-port `IOTSs`. We have that i* **ioco** *s if and only if i* **p-dioco** *s.*

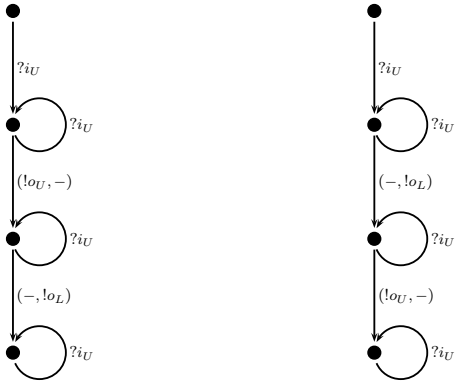


Fig. 8 Processes s_1 (left) and i_1 (right).

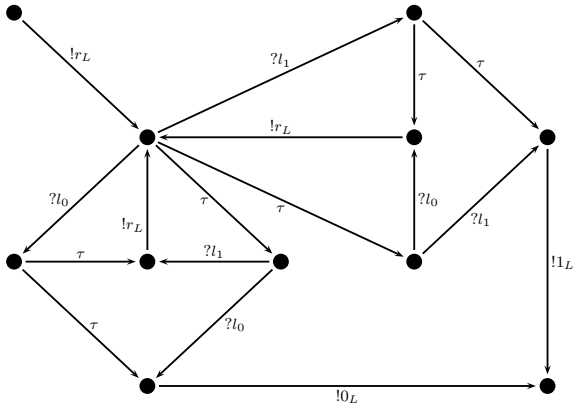


Fig. 9 Projection on port l of the majority voting protocol.

In order to provide an alternative characterisation of the **p-dioco** relation, we will show that **p-dioco** corresponds to comparing the *projections* of i and s using **ioco**. First, we formally introduce the notion of projection of a process.

Definition 8 Let $s = (Q, I, O, T, q_{in})$ be an `IOTS` and $p \in \mathcal{P}(s)$. We denote by $s\rceil_p$ the process $(Q, I_p, O_p, T', q_{in})$ formed by replacing in s all inputs that do not occur at p by τ and removing from the output tuples all outputs that do not occur at p. Thus, for all transition $q \xrightarrow{a} q' \in T$ we have that:

1. If $a = (!o_1, \ldots, !o_m)$, with $!o_p \neq -$, then $q \xrightarrow{!o_p} q' \in T'$.
2. If $a = (!o_1, \ldots, !o_m)$, with $!o_p = -$, then $q \xrightarrow{\tau} q' \in T'$.
3. If $a \in I_p \cup \{\delta\}$ then $q \xrightarrow{a} q' \in T'$.
4. If $a \in I_{p'}$, for some $p' \neq p$, then $q \xrightarrow{\tau} q' \in T'$.

Example 3 Figure 9 shows the projection on port L of the running example. Since the information concerning port U has been removed, the resulting protocol has some behaviours that do not seem compliant with the original protocol. For example, after receiving the input $?l_0$ it will non-deterministically decide either to output $!0_L$ at port L, concluding the process, or to output $!r_L$ at port L, reinitiating the process. Since it is not possible to see what was received from channel U, the decision is, apparently, non-deterministic. In conclusion, by considering projections, we are losing a lot of information about the causality relations in the protocol, but projections are the right tool to characterise the **p-dioco** relation, as shown in the forthcoming Proposition 6.

The next result relates the projected traces of a process and the traces of a projected process.

Proposition 5 *Let s be an `IOTS` and $p \in \mathcal{P}(s)$ be a port. We have that $\mathcal{T}r^*(s\rceil_p) = \pi_p(\mathcal{T}r^*(s))$.*

Proof We prove a result equivalent to the stated one: $\sigma_p \in \mathcal{T}r^*(s\rceil_p)$ if and only if $\sigma_p \in \pi_p(\mathcal{T}r^*(s))$. The proof will be carried out by induction on the length of σ_p. Clearly the base case, that is ϵ, holds. We therefore assume that the result holds for all elements of Act_p^* of length less than k, for some $k > 0$, and let σ_p be an arbitrary trace of length k. Thus, $\sigma_p = a\sigma_p'$ for some $a \in \mathcal{A}ct_p$.

First, let us assume that $a\sigma_p' \in \mathcal{T}r^*(s\rceil_p)$ and so there exists some $\sigma_1 \in \mathcal{A}ct^*$, with $\pi_p(\sigma_1) = a$, and s', with $s \stackrel{\sigma_1}{\Longrightarrow} s'$, such that $\sigma_p' \in \mathcal{T}r^*(s'\rceil_p)$. By the inductive hypothesis, $\sigma_p' \in \pi_p(\mathcal{T}r^*(s'))$ and then $a\sigma_p' \in \pi_p(\mathcal{T}r^*(s))$ as required.

Now, let us assume that $a\sigma_p' \in \pi_p(\mathcal{T}r^*(s))$ and so there exists some $\sigma_1 \in \mathcal{A}ct^*$, with $\pi_p(\sigma_1) = a$, and s', with $s \stackrel{\sigma_1}{\Longrightarrow} s'$, such that $\sigma_p' \in \pi_p(\mathcal{T}r^*(s'))$. By the inductive hypothesis, $\sigma_p' \in \mathcal{T}r^*(s'\rceil_p)$ and then $a\sigma_p' \in \mathcal{T}r^*(s\rceil_p)$ as required.

We can now characterise the relation **p-dioco** in terms of the **ioco** relation.

Proposition 6 *Let i, s be `IOTSs` with port set $\mathcal{P}$. We have that i **p-dioco** s if and only if for all $p \in \mathcal{P}$ we have $i\rceil_p$ **ioco** $s\rceil_p$.*

Proof First, let us note that, by applying Proposition 5, $\sigma_p \in \mathcal{T}r^*(s\rceil_p)$ if and only if there exists $\sigma \in \mathcal{T}r^*(s)$ such that $\pi_p(\sigma) = \sigma_p$.

We have that i **p-dioco** s if and only if for every $\sigma \in \mathcal{T}r^*(i)$ and $p \in \mathcal{P}$ there exists some trace $\sigma' \in \mathcal{T}r^*(s)$ such that $\pi_p(\sigma) = \pi_p(\sigma')$. This is the case if and only if $\pi_p(\mathcal{T}r^*(i)) \subseteq \pi_p(\mathcal{T}r^*(s))$. By Proposition 5, this holds if and only if $\mathcal{T}r^*(i\rceil_p) \subseteq \mathcal{T}r^*(s\rceil_p)$ and, by Proposition 2, this is the case if and only if $i\rceil_p$ **ioco** $s\rceil_p$ as required.

The implementation relation **p-dioco** corresponds to a situation in which the agent at one port has no knowledge of what is happening at the other ports. However, if one independent agent can receive information regarding observations made at different ports, then it is necessary to define a stronger implementation relation.

4 A stronger implementation relation

Under **p-dioco** the separate agents are independent and so two agents can compare their observations with different traces of the specification. Let us consider again the situation in which s is a process in which the input of $?i_U$ at U is either followed by $(!o_U, !o_L)$ or by $(!o'_U, !o'_L)$. We saw that under **p-dioco** it is acceptable for the SUT to produce the trace $?i_U(!o_U, !o'_L)$ since at each port the projection of $?i_U(!o_U, !o'_L)$ is a projection of a trace in the specification. However, the observations at U and L are projections of different traces of the specification and $?i_U(!o_U, !o'_L)$ is not equivalent to a trace of the specification under $\sim$. As a result, we can declare a failure if the observations made at the ports are brought together later. Thus, if the observations might be brought together after testing then we require a different implementation relation. In this section we define a new implementation relation **dioco**. We will see that this is a stronger version of the implementation relation defined in [25], which we will call **dioco**$_\delta$.

Definition 9 Let i, s be `IOTSs`. We write i **dioco**$_\delta$ s if for every (finite) trace $\sigma \in \mathcal{T}r^*(i)$ such that $\sigma\delta \in \mathcal{T}r^*(i)$, there exists a trace $\sigma' \in \mathcal{T}r^*(s)$ such that $\sigma' \sim \sigma$.

The implementation relation **dioco**$_\delta$ only considers traces that end with quiescence which is clearly a limitation. The motivation for this restriction is that in order to capture the idea underlying the new relation, it is necessary to compare sets of local traces with global traces. In addition, it must be ensured that the local traces are all projections of the same (unknown) global trace of the implementation. If this trace ends in quiescence then it is possible to check the appropriateness of the projections. In addition, the testers can choose to stop testing when the SUT is in a quiescent state.

Let us consider now a specification s that has a single state and only one transition, which has output $(!o_U, -)$. Further, let us suppose that an implementation i also has one state and has one transition, with output $(-, !o_L)$. Clearly, i should not conform to s and the tester at L can deduce this as soon as it observes the output $!o_L$. However, i and s have no quiescent traces and so we have that i conforms to s under **dioco**$_\delta$.

In this paper we strengthen **dioco**$_\delta$ by considering infinite traces. However, we have to be careful with the considered infinite traces. Let s be a specification in which we have output $(-, !o_L)$ followed by output $(!o_U, -)$ with self-loops with inputs $?i_U$ and $?i_L$ added to make this specification input-enabled. One of the infinite traces that s can perform is $(-, !o_L)$ followed by an infinite number of inputs at L. This is not a trace of the process s' obtained by switching outputs $(-, !o_L)$ and $(!o_U, -)$ in s. Similarly, $(!o_U, -)$ followed by an infinite number of inputs at U is an infinite trace of s' but not of s. However, we would not expect to be able to distinguish between s and s' since we cannot distinguish between $(-, !o_L)$ followed by $(!o_U, -)$ and $(!o_U, -)$ followed by $(-, !o_L)$ even if we add inputs. The problem is that by considering a trace such as $(!o_U, -)$ followed by an infinite number of inputs at U we effectively allow a tester (at U) to block an output at L by supplying an infinite number of inputs. This is not appropriate since the environment should not be able to block output.

In order to avoid the use of an infinite number of inputs to block output, we restrict our attention to possibly infinite traces that contain only a finite number of inputs. Let us note that we place no bound on the number of outputs. In particular, any behaviour that can occur as a result of a finite sequence of transitions is admissible. We also introduce the predicate **fail**(s, σ), that will be used in Section 7, that determines if the sequence σ can be produced, up to the $\sim$ relation, by the `IOTS` s.

Definition 10 Let i be an `IOTS` and $\sigma \in \mathcal{T}r^\omega(i)$. We say that σ is a *complete run of* i if it contains only a finite number of inputs. The set of complete runs of i is denoted by $\mathcal{R}(i)$.

Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. We write **fail**(i, σ) if there does not exist a complete run $\sigma' \in \mathcal{R}(i)$ such that for all $p \in \mathcal{P}$, $\pi_p(\sigma) \in pre(\pi_p(\sigma'))$.

By Proposition 1, every finite trace of i can be extended to form a complete run of i. It is straightforward to define **ioco** in terms of complete runs if s is input-enabled. The next result follows from Proposition 2.

Proposition 7 *Let i, s be `IOTSs`. We have that i* **ioco** *s if and only if $\mathcal{T}r^\omega(i) \subseteq \mathcal{T}r^\omega(s)$.*

Taking as initial step Proposition 7, we can show the relation between the complete runs of implementations and specifications related under **ioco**.

Proposition 8 *Let i, s be `IOTSs`. We have that i* **ioco** *s if and only if $\mathcal{R}(i) \subseteq \mathcal{R}(s)$.*

Proof First, let us assume i **ioco** s. For all sequence $\sigma \in \mathcal{R}(i)$, by definition, we have $\sigma \in \mathcal{T}r^{\omega}(i)$. This implies by Proposition 7 that $\sigma \in \mathcal{T}r^{\omega}(s)$, which implies by definition that $\sigma \in \mathcal{R}(s)$.

Now let us assume that $\mathcal{R}(i) \subseteq \mathcal{R}(s)$. Let us consider some $\sigma \in \mathcal{T}r^{*}(s)$ and $a \in \mathbf{out}(i \textbf{ after } \sigma)$. Then $\sigma a \in \mathcal{T}r^{*}(i)$. By Proposition 1, we know that an element of $\mathcal{R}(i)$ has σa as a prefix. Thus, an element of $\mathcal{R}(s)$ has σa as a prefix. It therefore follows that $a \in \mathbf{out}(s \textbf{ after } \sigma)$ as required.

We now define the novel **dioco** implementation relation.

Definition 11 Let i, s be IOTSs. We write i **dioco** s if and only if for all $\sigma \in \mathcal{R}(i)$ there exists some $\sigma' \in \mathcal{R}(s)$ such that $\sigma \sim \sigma'$.

Under the conditions considered in this paper, the implementation relation **dioco** is strictly stronger than $\mathbf{dioco}_{\delta}$.

Proposition 9 *For every IOTSs i and s, i* **dioco** *s implies i* $\mathbf{dioco}_{\delta}$ *s. However, there exist systems for which i* $\mathbf{dioco}_{\delta}$ *s but not i* **dioco** *s.*

Proof The first part follows from observing that for every trace σ of i that ends in quiescence, the trace formed by following σ by an infinite number of δ actions is an element of $\mathcal{R}(i)$ since it has a finite number of inputs. In order to see the second part, let us consider an implementation i that has only two transitions, a self-loop with output $!y$ and a self-loop with input $?x$, and a specification that also has two transitions, a self-loop with output $!y'$, with $!y' \neq !y$, and a self-loop with input $?x$. Then, i and s have no traces that end in quiescence and so i $\mathbf{dioco}_{\delta}$ s but clearly i **dioco** s does not hold.

The presence of physically distributed interfaces reduces the ability of testers to observe behaviours of the SUT. As one would expect, **dioco** is strictly weaker than **ioco** since all observations that can be made locally can also be made globally.

Proposition 10 *For every IOTSs i and s, i* **ioco** *s implies i* **dioco** *s. However, there exist systems for which i* **dioco** *s but not i* **ioco** *s.*

Proof First, if i **ioco** s and $\sigma \in \mathcal{R}(i)$ then, by Proposition 8, $\sigma \in \mathcal{R}(s)$. Therefore, i **dioco** s.

Now, let us consider the processes s_2 and i_2 depicted in Figure 10. It is clear that these processes are incomparable under **ioco** since if an output is produced then its value 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 under **dioco**.

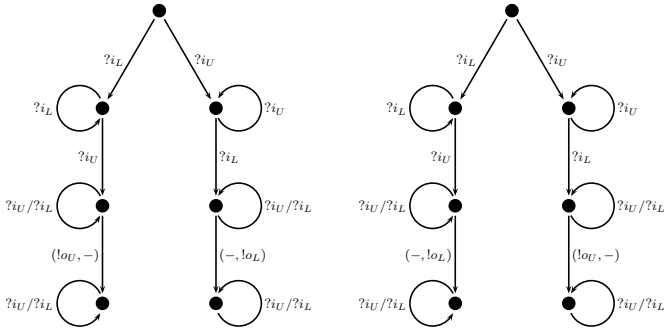


Fig. 10 Processes s_2 (left) and i_2 (right) that are not related under **ioco**.

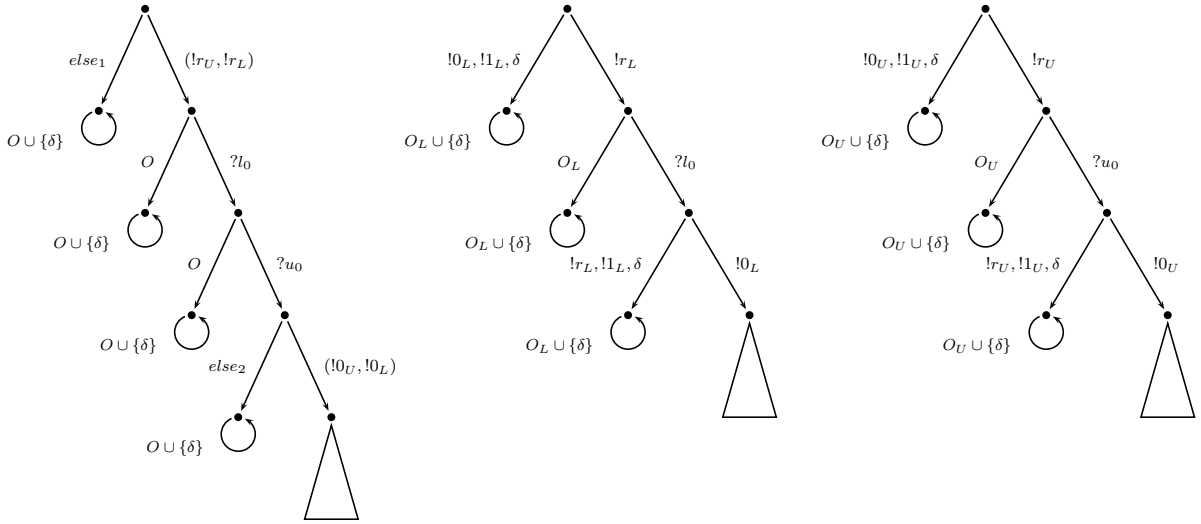
Since **dioco** adapts **ioco** to the case where there are multiple ports, we expect **dioco** to be equivalent to **ioco** if there is only one port. Interestingly, this equivalence does not hold for the previous implementation relation $\mathbf{dioco}_{\delta}$: For single-port i and s it is possible that i $\mathbf{dioco}_{\delta}$ s but not i **ioco** s. In fact, the same counterexample used in the proof of Proposition 9 can be used here. However, the new implementation relation **dioco** introduced in this paper fulfills this interesting and desirable property. In fact, one of the important results of this paper is the introduction of an extension of **ioco** that coincides with it in the case of single-port IOTSs. The result follows from the definition of **dioco** and Proposition 8.

Proposition 11 *Let i, s be single-port IOTSs. We have that i* **ioco** *s if and only if i* **dioco** *s.*

We conclude this section by relating the implementation relations that we have defined so far. It is trivial to check that **dioco** implies **p-dioco** as the following result states.

Proposition 12 *Let i, s be IOTSs. We have that i* **dioco** *s implies i* **p-dioco** *s.*

However, there is no relation between **p-dioco** and $\mathbf{dioco}_{\delta}$. The example used in the proof of Proposition 9 shows that i $\mathbf{dioco}_{\delta}$ s does not imply i **p-dioco** s while the example given in the beginning of this section shows that i **p-dioco** s does not imply i $\mathbf{dioco}_{\delta}$ s. We would expect $\mathbf{dioco}_{\delta}$ to be strictly stronger than **p-dioco** which motivated the development of **dioco**.



where $else_1 = O \setminus \{(!r_U, !r_L)\} \cup \{\delta\}$ and $else_2 = O \setminus \{(!0_U, !0_L)\} \cup \{\delta\}$.

Fig. 11 Global (left) and local (center and right) test cases.

5 Test cases for the distributed test architecture

In order to define testing in the distributed test architecture it is necessary to adapt the standard definition of a test case. In this paper, a *global test case* t for a process s is an IOTS with the same input and output sets (including δ) as s. Thus, we can use for test cases the notation introduced for IOTSs in Section 2. As usual, processes and tests synchronise on common values. Given the fact that 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 introduce the notion of a *local test case* that, in contrast to global test cases, contains a process for each port rather than a single process. In the next definition we first introduce notation to define simple tests that cannot send any input.

Definition 12 Let I and O be input and output sets. Let $\perp^{I,O} = (\{q\}, I, O, T, q)$ be the IOTS such that $T = \{(q, !o, q) | !o \in O \cup \{\delta\}\}$ and thus whose finite traces are all the elements of $(O \cup \{\delta\})^*$. Similarly, let $\perp_p^{I,O} = (\{q\}, I, O_p, T, q)$ be the IOTS such that $T = \{(q, !o, q) | !o \in O_p \cup \{\delta\}\}$ and thus whose finite traces are all the elements of $(O_p \cup \{\delta\})^*$. We will omit I and O when they can be inferred from the context.

Let $s = (Q, I, O, T, q_{in})$ be an IOTS. A *global test case* for s is an IOTS $t = (Q', I, O \cup \{\delta\}, T', q'_{in})$, such that the following restrictions hold:

- t is *output-enabled*, that is, $t \stackrel{\sigma}{\Longrightarrow} t'$, for some sequence σ, implies $t' \stackrel{!o}{\Longrightarrow}$ for all $!o \in O \cup \{\delta\}$,
- t has no transitions labelled by τ,
- $\mathcal{P}(t) = \mathcal{P}(s)$,
- Q' is finite, and
- the graph induced by T is acyclic except for those loops created by occurrences of $\perp$ and $\perp_p$.

We say that t is *deterministic* if for every sequence σ, t **after** σ contains at most one process and t does not have a state with more than one outgoing transition labelled with an input.

A *local test case* for s is a tuple $(t_1, \ldots, t_m)$ of *local testers* in which for all $p \in \mathcal{P}(s)$, t_p is a deterministic (global) test case with input set I_p and output set $O_p \cup \{\delta\}$.

A local test case is thus a collection of *global* test cases, where each of these test cases can observe only at one port. The notion of *output-enabled* expresses the idea that in each state a test case must be able to accept any output from the SUT. As usual, during the rest of the paper we only consider (local or global) deterministic test cases. Figure 11 shows a global test case (left) and a local test case (center and right) for the specification introduced in Example 1. We next give some notation that will be used in Section 7. An assignment such as $q :=\perp$ denotes that the state q has a self-loop transition for each output, including δ.

Definition 13 Let $s = (Q, I, O, T, q_{in})$ be an IOTS and $q \in Q$. We consider that $q :=\perp$ adds to T the following transitions $\{(q, !o, q) | !o \in O \cup \{\delta\}\}$.

In order to apply a global test case to a system, we simply consider synchronisation on common actions. In

addition, we have to define how a local test case evolves (global test cases are IOTSs and thus follow the same rules as usual systems).

Definition 14 Let $i = (Q, I, O, T, q_{in})$ be an IOTS such that $\mathcal{P}(i) = \{1, \ldots, m\}$ and let $t = (t_1, \ldots, t_m)$ be a local test case for i. We write $t \stackrel{\sigma}{\Longrightarrow} t'$, for $t' = (t'_1, t'_2, \ldots, t'_m)$ and $\sigma \in \mathcal{A}ct$, if for all $1 \leq p \leq m$, $t_p \stackrel{\pi_p(\sigma)}{\Longrightarrow} t'_p$. For $\sigma = a_1, a_2, \ldots \in \mathcal{A}ct^* \cup \mathcal{A}ct^\omega$ we write $t \stackrel{\sigma}{\Longrightarrow}$ if there exists $t', t'', \ldots$ such that $t \stackrel{a_1}{\Longrightarrow} t'$, $t' \stackrel{a_2}{\Longrightarrow} t'', \ldots$ In this case we say that σ is a *trace* of t. Again, $\mathcal{T}r(t)$ denotes the set of (finite and infinite) traces that can be produced by t.

Let t be a (local or global) test case for i. We say that σ is a *trace* of the composition of i and t if $i \stackrel{\sigma}{\Longrightarrow}$ and $t \stackrel{\sigma}{\Longrightarrow}$. We let $\mathcal{T}r(i, t)$ denote the set of traces that can be observed when testing i with t.

Let us note that by definition all elements of $\mathcal{T}r(t)$ and $\mathcal{T}r(i, t)$ contain only a finite number of inputs. The next result easily follows from the fact that in testing a system i with a local test case t the SUT synchronises on visible actions with the local testers.

Proposition 13 *Let i be an IOTS and t be a local test case. Then, $\mathcal{T}r(i) \cap \mathcal{T}r(t) = \mathcal{T}r(i, t)$.*

In testing we only observe local traces. As a result we could say that σ is a trace of the composition of i and t if $i \stackrel{\sigma}{\Longrightarrow}$ and there exists σ' such that $t \stackrel{\sigma'}{\Longrightarrow}$ and $\sigma \sim \sigma'$. We chose Definition 14 since it leads to a smaller set of traces and simplifies some of the analysis, but the main results in this paper do not depend on which of these two choices we make. As a result of Definition 14 we obtain the following.

Proposition 14 *Let i be an IOTS and t be a global test case. Then, $\mathcal{T}r(i) \cap \mathcal{T}r(t) = \mathcal{T}r(i, t)$.*

The consideration of a global test case t synchronising with a process i effectively requires global traces to be observed by the global test case. To see this let us consider a process i in which input $?i_U$ can be followed by output $(-, !o_L)$ and a global test case t that can apply input $?i_U$ after output $(-, !o_L)$. Under Definition 14 we have that $\mathcal{T}r(i, t) = \{\epsilon\}$. However, if we were to attempt to implement t using local testers then the tester at U would not know whether $!o_L$ had been observed and so would supply $?i_U$. As a result the global trace $?i_U(-, !o_L)$ could occur. The motivation for Definition 14 is to use a global test case to define a set of possible behaviours that we would like to test, but we will not actually use global test cases in testing: Instead we will *approximate* a global test case (a desired way of testing the SUT) using a set of local test cases.

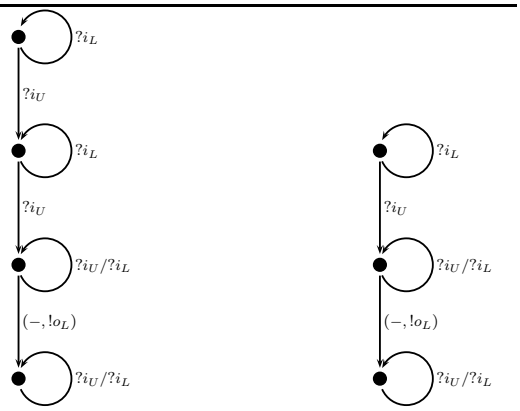


Fig. 12 Testing can be non-monotonic.

It is interesting to note that test effectiveness is not *monotonic* in the distributed test architecture, that is, the application of an input sequence can reveal a failure but we could extend this input sequence and lose the ability to find a failure. Let us consider the specification s depicted in Figure 12, left, and the implementation i depicted in Figure 12, right. If the local tester at U applies $?i_U$ and then waits for output and the local tester at L just waits for 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 waits for output and the local tester at L just waits for output, then we do not detect a failure since testing will observe $?i_U?i_U$ at U, as expected, and $!o_L$ at L also as expected.

In contrast with *classical* testing approaches, test cases do not include a notion of pass/fail state. Usually, the decision about passing/failing a test run will be taken by putting together all the observations obtained during the application of the test case in order to reach a verdict. However, local testers do not have enough information to independently reach a verdict. We now discuss this issue in greater detail.

Local traces are observed at each port. On the basis of these observations, a verdict has to be reached: The verdict being *pass* if the observations are consistent with the specification and *fail* otherwise. For the **p-dioco** implementation relation this task is relatively straightforward since, due to Proposition 6, it is enough to compare a local trace at a port p with $s\rceil_p$ under the **ioco** implementation relation. So, the presence of multiple ports makes no difference for **p-dioco** since the verdicts produced by local testers can be combined in the obvious way (the overall verdict is *pass* if and only if all local testers return the verdict *pass*).

On the contrary, the implementation relation **dioco** compares infinite traces of the SUT with infinite traces of the specification. In testing, however, the final verdict

must be reached based only on finite traces. Essentially, the verdict *fail* must be assigned to a set of local traces whenever it necessarily follows that the SUT is not an acceptable implementation.

By definition, if δ is observed at one port then it is also observed at all ports. This property allows us to identify local traces that correspond to the same global trace. Hence, we start by considering quiescent traces.

Definition 15 Let i, s be IOTSs.

1. A sequence σ is a *quiescent run* for i if $\sigma\delta \in \mathcal{T}r(i)$.
2. We say that i **passes** the quiescent run σ for s if there exists some $\sigma' \in \mathcal{T}r(s)$ such that $\sigma' \sim \sigma$. Otherwise, i **fails** the quiescent run σ.
3. A sequence σ is a *run* for i if $\sigma \in \mathcal{R}(i)$ or σ is a quiescent run for i.
4. We say that i **passes** the run σ for s if there exists some $\sigma' \in \mathcal{T}r(s)$ such that $\sigma' \sim \sigma$. Otherwise, i **fails** the run σ.
5. Let t be a (local or global) test case. We say that i **passes** t for s if for every run $\sigma \in \mathcal{T}r(i,t)$, i **passes** σ for s. Otherwise, we say that i **fails** t for s.

Let us remark that in order to consider this notion of *passing* test cases in the non-distributed architecture it is enough to replace in the previous definition the occurrences of $\sim$ by $=$. The next result follows immediately from Definition 15.

Proposition 15 *Let i, s be IOTSs. Then, i* **dioco** *s if and only if for every run $\sigma \in \mathcal{T}r(i)$, i* **passes** *σ for s.*

Propositions 14 and 15 imply that the set of global test cases has the same distinguishing power as **dioco**. In Section 7 we present an algorithm to produce a set of *relevant* test cases.

Proposition 16 *Let i, s be IOTSs. Then, i* **dioco** *s if and only if for every global test case t, i* **passes** *t for s.*

Proof First let us assume that i **dioco** s and let t be a global test case. We are required to prove that i **passes** t for s. Let $\sigma \in \mathcal{T}r(i,t)$ be a run of i with t. By definition, σ is either an infinite element of $\mathcal{R}(i)$ or σ is a finite element of $\mathcal{T}r(i)$ that ends in δ. In the first case, since i **dioco** s, there exists $\sigma' \in \mathcal{R}(s)$ such that $\sigma' \sim \sigma$ and so i **passes** σ as required. In the second case, σ can be extended with an infinite number of δ actions to form a trace $\sigma_1 \in \mathcal{R}(i)$. Thus, there exists some $\sigma'_1 \in \mathcal{R}(s)$ such that $\sigma'_1 \sim \sigma_1$. By the definition of $\sim$, the sequence σ'_1 has a prefix $\sigma' \sim \sigma$. Thus, i passes σ as required.

Now let us assume that for every global test case t, i **passes** t for s and let σ be an element of $\mathcal{R}(i)$. It is sufficient to prove that there exists $\sigma' \sim \sigma$ such that $\sigma' \in \mathcal{R}(s)$. Let σ_1 denote the shortest prefix of σ that contains all of the inputs of σ; clearly some such σ_1 exists. Let us consider a test case t such that σ_1 takes t to state $\perp$. Then $\sigma \in \mathcal{T}r(t)$ and so $\sigma \in \mathcal{T}r(i,t)$ is a run for i. Thus, since i **passes** t, there is some $\sigma' \in \mathcal{T}r(s)$ such that $\sigma' \sim \sigma$. Finally, since $\sigma' \sim \sigma$, the sequence σ' has a finite number of inputs and so $\sigma' \in \mathcal{R}(s)$ as required.

It is relatively straightforward to determine whether an implementation passes a quiescent run. In general, however, the notion of passing a run is defined in terms of infinite traces while, in our case, only finite traces are observed at each port. The following result gives a condition under which we can conclude that the implementation has a failing run from observing a finite trace.

Proposition 17 *Let i, s be IOTSs with port set $\mathcal{P}$ and $\sigma \in \mathcal{T}r^*(i)$. If there does not exist $\sigma' \in \mathcal{T}r^\omega(s)$ such that $\pi_p(\sigma) \in pre(\pi_p(\sigma'))$ for all $p \in \mathcal{P}$, then i has a failing run for s.*

Finally, the following result is a special case of the previous one.

Proposition 18 *Let i, s be IOTSs with port set $\mathcal{P}$ and $\sigma \in \mathcal{T}r^*(i)$. If there is a port $p \in \mathcal{P}$ such that there does not exist $\sigma' \in \mathcal{T}r^\omega(s)$ such that $\pi_p(\sigma) \in pre(\pi_p(\sigma'))$, then i has a failing run for s.*

We conclude this section with a discussion on the omission of verdicts in states of the test cases. Since we can assign a verdict based on the specification, it is not necessary to include verdicts in test cases and this simplifies their description. The assignment of verdicts to states in the non-distributed architecture also uses specifications: A pass/fail state means that the specification can/cannot exercise the trace leading to that state of the test case. In addition, the verdict for a test case depends on observations made by the local testers which for **dioco** is not simply a case of combining verdicts made by these local testers. As a result, the traditional approach of associating verdicts with states of the test case does not seem to be appropriate in our framework.

6 Deterministic and controllable test cases

Test objectives are usually stated during the specification phase. As a result, it is natural to generate an initial global test case that achieves a given test objective. The challenge is then to produce local testers that implement a given global test case. In this section we

present a restricted family of global test cases that can be implemented using local test cases without introducing additional nondeterminism into testing. Then, we give an algorithm to decide whether a test case belongs to the family.

6.1 Formal definitions

We continue with the restriction to study only deterministic test cases: When testing i with a local test case $(t_1, \ldots, t_m)$, t_p is deterministic for all $p \in \mathcal{P}(i)$. However, the local test case $(t_1, \ldots, t_m)$ can still be nondeterministic. For example, let us consider t_{p_1} and t_{p_2}, with $p_1 \neq p_2$, that both start by sending an input to the implementation: The order in which the implementation receives these inputs from t_{p_1} and t_{p_2} is not predictable. Thus, even if all the local testers are deterministic, nondeterminism occurs in $(t_1, \ldots, t_m)$ if there are ports $p \neq q$ such that at some point in a test both t_p and t_q can provide an input. More generally, nondeterminism can be caused by a local test case that contains deterministic local testers if there can be a race between an event at one port q and input at another port p. We now explore deterministic local test cases before stating what it means for a global test case to be *controllable*. Let us note that issues relating to controllability have been studied in the context of testing from a deterministic finite state machine [49,13,6,11,9,27,46,52,53,28,10].

Let us consider a local test case $(t_1, \ldots, t_m)$ such that t_p and $t_{p'}$ can both send input to the SUT for some $p \neq p'$. Therefore, there exists a trace σ_p at p after which t_p will send input and a trace $\sigma_{p'}$ at p' after which $t_{p'}$ sends input. Thus, there will be nondeterminism if the SUT can produce some trace $\sigma' \in \mathcal{A}ct^*$ with $\pi_p(\sigma') = \sigma_p$ and $\pi_{p'}(\sigma') = \sigma_{p'}$. As a result, it is reasonable to require the local test case to be deterministic for all traces of the specification.

Definition 16 Let $s = (Q, I, O, T, q_{in})$ be an IOTS and t be a local test case. We say that t is *deterministic for* s if there do not exist port $p \in \mathcal{P}$, $\sigma_1, \sigma_2 \in \mathcal{A}ct^*$, with $\sigma_2 \sim \sigma_1$, $a \in \mathcal{A}ct \setminus \mathcal{A}ct_p$, and $?i_p \in I_p$ such that $\sigma_1?i_p, \sigma_2 a \in \mathcal{T}r(s,t)$.

This definition captures that testing s with a local test case t is deterministic whenever if t is interacting with s, there cannot be a race between an input at a port p and an event (input or output) a at another port $q \neq p$. Let us reiterate that, by definition, the local testers in a local test case must themselves be deterministic.

It is known that it is not sufficient for a global test case to be deterministic when testing in the distributed test architecture [24]. The reason is that each local tester must decide when to apply an input on the basis of its local observations. Let us consider, for example, a global test case in which p' must apply input $?i_{p'}$ after $?i_p!o_p$: The tester at p' does not observe the actions $?i_p$ or $?o_p$. Hence, this observer cannot know when to apply $?i_{p'}$ and there is a race between $?i_{p'}$ and both $!o_p$ and $?i_p$. The following notion generalises the definition of a global test case being *controllable* [24] to the case with more than two ports.

Definition 17 Let s be an IOTS. A global test case t is said to be *controllable* for s if there do not exist $\sigma_1, \sigma_2 \in \mathcal{T}r(s,t)$, port $p \in \mathcal{P}(s)$ with $\pi_p(\sigma_1) = \pi_p(\sigma_2)$ and an input $?i_p \in I_p$ such that $\sigma_1?i_p \in \mathcal{T}r(s,t)$ but $\sigma_2?i_p \notin \mathcal{T}r(s,t)$.

We restrict attention to global test cases since we are looking for conditions under which we can implement a global test case using local test cases. In addition, local test cases immediately satisfy the conditions of Definition 17 but a local test case might not be deterministic for s.

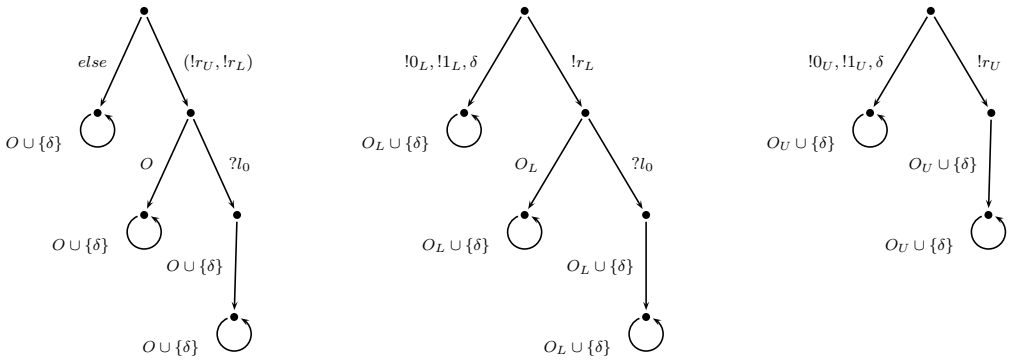
Example 4 The global test case depicted in Figure 11, left, is not controllable for the specification given in Figure 5 since the right order in which the inputs have to be applied is unknown. Technically, the sequences $(!r_U, !r_L)$ and $(!r_U, !r_L)?l_0$ produce the same projection on port U. However, by considering the input $?u_0$, the sequence $(!r_U, !r_L)?l_0?u_0$ can be generated by the composition of the test case and the specification while the sequence $(!r_U, !r_L)?u_0$ cannot be generated by this composition.

On the contrary, the global test case depicted in Figure 13, left, is controllable for the specification given in Figure 5.

We study now the problem of generating a local test case that implements a controllable global test case.

Definition 18 Let s be an IOTS, t be a global test case, and t_l be a local test case. We say that the local test case t_l *implements* the global test case t for specification s if the interaction between 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 all σ we have $\sigma \in \mathcal{T}r(s, t_l)$ if and only if $\sigma \in \mathcal{T}r(s,t)$.

Even though the notion of controllability is restrictive, it is important since it appropriately captures the idea of the local testers knowing when to apply input. This feature is clearly a highly desirable property and



where $else = O \setminus \{(!r_U, !r_L)\} \cup \{\delta\}$

Fig. 13 A controllable test case and its related local testers.

probably the main reason for almost all of the FSM work only considering controllable test sequences (see, for example, [49,13,14,6,40,11,50,27,29]). The problem is that while the notion of controllability is elegant in this sense, it is very restrictive and it is easy to construct models for which, for example, there are transitions that cannot be tested if only controllable tests are considered. Facing this situation, there are two scenarios:

- Controllable test cases: Test cases have nice properties but they are restrictive.
- Test cases are not controllable: The tests are more difficult to deal with and produce some counterintuitive results but are more general.

A good theory for testing in the distributed architecture must establish a trade off between the two. In practice, we would like to "test as much as possible" in a controllable manner. An intrinsic limitation is that there are properties of interest for which there is no controllable test case. In any case, it is desirable to use controllable test cases in testing. Therefore, in Section 6.3 we will show how to decide whether a global test case is controllable and we will then define a new implementation relation that corresponds to testing only with controllable test cases.

Efficiency, in particular of taking projections of a global test cases to form local test cases, is an important issue that we discuss in the next section. This issue is illustrated by the following situation.

1. A specification s in which the only possible transition is from the initial state to another state s' through output $(!o_U, -)$. In addition, all inputs are available in both states, producing a loop.
2. A global test case t that after $(!o_U, -)$ can apply input $?i_L$ at port L and that after $(!o'_U, -)$ can apply input $?i'_L$ at port L.

When t and s synchronise, t cannot send input $?i'_L$ since it cannot receive output $(!o'_U, -)$ from s. However, the projections of t into the resultant local tester at port L can send input $?i'_L$. This situation happened because there are states of t that cannot be exercised when t is synchronised with s. We now define a condition to guarantee that this situation cannot happen.

Definition 19 Let s be an IOTS. A global test case t is said to be *reduced* for s if there do not exist a trace $\sigma \in \mathcal{A}ct^*$ and an input $?i$ such that $\sigma?i \in \mathcal{T}r(t) \setminus \mathcal{T}r(s)$.

Clearly, if a global test case t is not reduced for s, when testing from s we can remove parts of it in order to produce a reduced global test case t' such that $\mathcal{T}r(s,t) = \mathcal{T}r(s,t')$ and so we lose nothing in restricting attention to reduced global test cases.

Let us note that a global test case might not be reduced even if the specifications is input-enabled. To see this, it is sufficient to consider a trace σ that cannot be performed by s. Even if s is input enabled, a test case t is not reduced if $\mathcal{T}r^*(t)$ contains a trace formed by following σ by an input. In addition, a controllable test case might not be reduced. Let us consider, for example, a global test case t in which input $?i_U$ occurs after $(-, !o_L)$ but s cannot produce $(-, !o_L)$ in its initial state. The test case t is controllable for s but is not reduced.

6.2 Obtaining local test cases from global test cases

In the appendix of the paper we provide a method to produce local testers by taking projections of a controllable global test case t. Intuitively, given a global test case t, the local test case $t_l = (det(t\rceil_1), det(t\rceil_2), \ldots, det(t\rceil_m))$ is constructed by first projecting the actions of t into each port and then transforming each of the resulting graphs into deterministic ones.

The following results state that if the method is applied to a reduced controllable global test case t, then the local test case implements t.

Proposition 19 *Let s be an IOTS with port set $\mathcal{P}(s) = \{1, \ldots, m\}$ and t be a controllable reduced global test case for s. The test case $t_l = (det(t\rceil_1), det(t\rceil_2), \ldots, det(t\rceil_m))$ implements t for s.*

Proof It is enough to prove that $\mathcal{T}r(s,t) = \mathcal{T}r(s,t_l)$. Clearly, $\mathcal{T}r(s,t) \subseteq \mathcal{T}r(s,t_l)$ so it is sufficient to prove that $\mathcal{T}r(s,t_l) \subseteq \mathcal{T}r(s,t)$.

By contradiction, let us assume that $\mathcal{T}r(s,t_l) \not\subseteq \mathcal{T}r(s,t)$, so $\mathcal{T}r(s,t_l) \setminus \mathcal{T}r(s,t) \neq \emptyset$. Let σ be a shortest prefix of an element of $\mathcal{T}r(s,t_l)$ that is not a prefix of an element of $\mathcal{T}r(s,t)$. Clearly $\sigma \neq \epsilon$. Furthermore, since a test case can accept any output, the last element of σ must be an input, that is, $\sigma = \sigma'?i_p$ for some port p and $?i_p \in I_p$.

Let $t_{lp} = det(t\rceil_p)$. Since $\sigma'?i_p$ is a prefix of an element of $\mathcal{T}r(s,t_l)$, $\pi_p(\sigma')?i_p$ is the prefix of some sequence in $\mathcal{T}r(t_{lp})$ and so there must exist some σ'' such that $\sigma''?i_p$ is a prefix of a trace in $\mathcal{T}r(t)$ and $\pi_p(\sigma'') = \pi_p(\sigma')$. Moreover, since t is reduced for s, $\sigma''?i_p$ is a prefix of a trace in $\mathcal{T}r(s,t)$. But since $\sigma'?i_p$ is not a prefix of an element of $\mathcal{T}r(s,t)$, this fact contradicts t being controllable for s.

The following result is an immediate corollary of Proposition 19, Definition 16, and the global test case considered being deterministic.

Corollary 1 *Let s be an IOTS with port set $\mathcal{P}(s) = \{1, \ldots, m\}$ and t be a controllable reduced global test case for s. Then, $t_l = (det(t\rceil_1), det(t\rceil_2), \ldots, det(t\rceil_m))$ is deterministic for s.*

6.3 Deciding whether a test case is controllable

Previous work [24] investigated the problem of deciding whether a test case is controllable, but this was restricted to processes with two ports and disallowing different outputs being sent at different ports at the same time. Under these conditions, the traces produced by a test case t with a specification s generated a set of Message Sequence Charts (MSCs). A condition CC3 from [1] regarding MSCs was then used, it being proved that the MSCs corresponding to $\mathcal{T}r(s,t)$ satisfy CC3 if and only if t is controllable for s.

MSCs do not consider messages that are sent at the same time. Therefore, MSCs cannot model $!o \in O$ containing output at more than one port. We could try to overcome this by choosing a fixed order in which to represent the values in $!o$, by ordering the ports and then ordering the individual outputs in the same way. For example, if $!o = (!o_1, \ldots, !o_m)$ then we could represent $!o$ by the sending of $!o_1!o_2 \ldots !o_m$ by the SUT. However, let us consider the situation in which a trace has as prefix $(!o_1, !o_2)?i_1$. The induced MSC would send the message $!o_1$ from the SUT to the tester at port 1 followed by $!o_2$ from the SUT to the tester at port 2, followed by the sending of $?i_1$ from the tester at port 1 to the SUT. This interaction appears to be uncontrollable (and therefore it fails condition CC3) as there should be another MSC in which the message $?i_1$ from the tester at port 1 to the SUT follows $!o_1$. The MSC corresponding to $!o_1?i_1!o_2$ would be an implied MSC. However, $(!o_1, !o_2)?i_1$ is controllable. The problem arises by separating the elements of a tuple creating the possibility of interaction between these outputs. Here, instead we define a new algorithm.

Algorithm 1
Input: Specification s and reduced global test case t.
Output: If t is controllable for s, output `True` else `False`.

{`Initialisation`}
Let N denote the set of nodes of t;
For every node $n \in N$ and port p, let n_p be the projection into port p of the trace of t reaching n;
{`Main loop`}
For all $n, n' \in N$, with $n \neq n'$, and $p \in \mathcal{P}$ do

- If $n_p = n'_p$ and there is an input $?i_p \in I_p$ such that $n \xrightarrow{?i_p}$ and $n' \not\xrightarrow{?i_p}$ then output `False` and terminate

output `True` and terminate.

Fig. 14 Deciding whether a global test case is controllable for a process.

Essentially, a global test case is a tree in which each leaf has a self loop for each output. Algorithm 1 in Figure 14 simply considers the nodes of t such that at least one of them can be followed by an input and determines whether any two of these nodes create a situation in which there is a controllability problem. Since t is reduced, all nodes of t can be reached when synchronising with s and the following result easily follows.

Theorem 1 *Let s be an IOTS and t be a reduced global test case. Algorithm 1 returns* `True` *for s and t if and only if t is controllable for s.*

The following result provides the worst case complexity of the algorithm.

Theorem 2 *Let s be an IOTS with m ports and t be a (deterministic and reduced) global test case with n nodes and maximum depth k. Algorithm 1 operates in time of $O(mn + n^2k)$.*

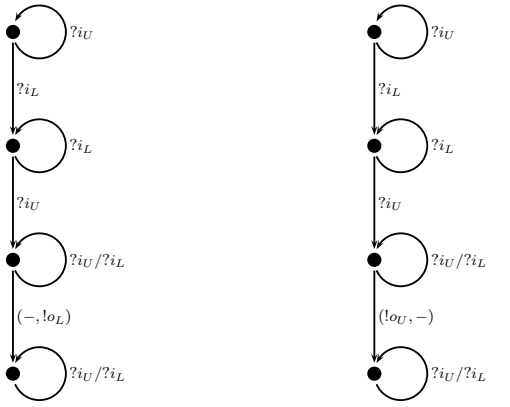


Fig. 15 Processes s_3 (left) and i_3 (right).

Proof The computation of the projections of the trace associated with a node n reached by an edge with label a from its parent n' can be achieved in constant time, assuming that the values for n' are available, if $a \in I_p$ for some $p \in \mathcal{P}$, since it is enough to extend the projection at p with a. If a is an output then it can be necessary to extend all of the projections and so this step can be performed in $O(m)$ time. Thus, starting from the root of the tree, the set of projections of traces associated with the different nodes can be computed in $O(mn)$. Finally, the loop contains at most $O(n^2)$ comparisons, because t is deterministic and hence every node has at most one outgoing input labelled edge. Each comparison takes at most time k and so the loop takes time of $O(n^2k)$.

6.4 Interactions that correspond to controllable test cases

By restricting interactions to controllable global test cases we obtain a new implementation relation, which is a generalisation of the implementation relation in [24]. The implementation relation in [24] was restricted to a type of `IOTSs` with only two ports and considered only traces that ended in quiescence.

Definition 20 Let i, s be `IOTSs`. We write i **c-dioco** s if for every controllable global test case t, i **passes** t for s.

We now investigate how **c-dioco** relates to **dioco** and thus also to **ioco**.

Proposition 20 *Let i, s be `IOTSs`. Then, i* **dioco** *s implies i* **c-dioco** *s. However, there are `IOTSs` i' and s' for which i'* **c-dioco** *s' but not i'* **dioco** *s'.*

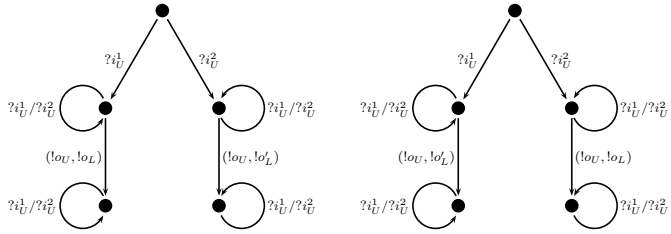


Fig. 16 Processes s_4 (left) and i_4 (right).

Proof The first part follows immediately from the definitions, with **c-dioco** restricting consideration to controllable global test cases.

For the second part, let us consider the processes $s' = s_3$ and $i' = i_3$ shown in Figure 15. It is clear that these processes are incomparable under **dioco**. However, all controllable global test cases for s' involve input at no more than one port and for each such test case neither process can produce output. It follows that i' **c-dioco** s'.

Proposition 21 *There exist `IOTSs` i and s such that i* **c-dioco** *s but not i* **p-dioco** *s. Furthermore, there exist `IOTSs` i' and s' such that i'* **p-dioco** *s' but not i'* **c-dioco** *s'.*

Proof First, let us consider again the processes s_3 and i_3 shown in Figure 15, which are comparable under **c-dioco** but not under **p-dioco**. Now, let us consider the processes s_4 and i_4 in Figure 16, which are equivalent under **p-dioco**. The test case that simply applies $?i_U^1$ and then observes output demonstrates that i_4 does not conform to s_4 under **c-dioco**.

We can now summarise the relationships between the four implementation relations presented in this paper as the transitive closure of the relation shown in Figure 17, where there is a sequence of arrows from one implementation relation r_1 to another implementation relation r_2 if the following holds: If $i\ r_1\ s$ then $i\ r_2\ s$ but the converse is not the case.

7 Test Generation

In this section we introduce a test derivation algorithm to generate minimal test suites for the **dioco** relation. These suites will fulfill two important properties. A test suite is *sound* if conforming implementations pass all the test cases in the suite. It is *complete* if non-conforming implementations fail some of these test cases.

We do not present a test generation algorithm for our notion of **ioco** since it is the same as that given

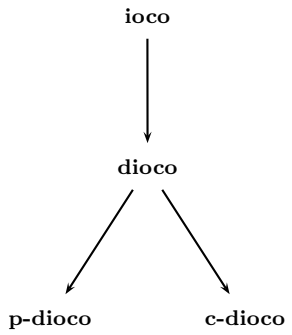


Fig. 17 Comparing the different implementation relations.

for **dioco** since the distinction between these two notions relates to the way in which test cases are applied to SUTs and not to the actual test suites themselves. Moreover, we do not present an algorithm to generate test cases for the **p-dioco** relation since this algorithm can be inferred from the **dioco** algorithm by projecting the results of test application on the different ports of the system. The algorithm that generates a complete test suite for **c-dioco** starts by considering the subset of controllable global test cases generated by the previously mentioned algorithm. Before we present our algorithm, let us note that, in our current framework, we cannot in general achieve *real* completeness of the test suite since, in general, the generated test suite will be infinite. But this is a common problem as the following simple example, adapted from [42], shows. Let us consider the following *specification*:

```
x in {0,1}
while true do begin
  read(x); write(x)
end
```

In order to test whether a given program conforms to this specification, the tester has to provide values and check the screen. If at a certain point the provided and observed values are different, then the tester knows that the implementation is not correct. However, if the tester has no information about the internal structure of the program, how many values must be exercised to ensure that the implementation is correct? For example, the tester might have the following *implementation*:

```
Program simple;
  begin
     for i:=1 to 10000000 do begin
        read(x); write(x)
     end;
     read(x); write('I am faulty');
     while true do begin
        read(x); write(x)
     end
  end.
```

In order to overcome this problem, some formal testing approaches assume that some information about the SUT is available. For example, it is very common to consider that the SUT has a known number of states and that it is *somehow* deterministic. In the previous example, if we assume that the implementation has a single state (let us note that the program `simple` has more than one state) and that it is deterministic, then there exist testing methods to determine whether the implementation is correct or faulty. Since in this paper we do not assume any additional information, in general, we will not be able to provide a conclusive verdict (in finite time). However, our methodology allows testing to ensure completeness *in the limit* of any SUT (that is, without assuming additional hypotheses). In this running example, we would be able to provide verdicts such as "the program works as expected if we provide n values or less." If n tends to infinite then, in the limit, we can prove the correctness of the SUT. Specifically, we obtain completeness in the limit by requiring that for all non-conforming SUT the algorithm will eventually produce a test case that will be failed by the SUT.

Algorithm 2 is given in Figure 18. It is *nondeterministic* since in each iterative step it can proceed in different ways. Each election generates a different global test case. We use $\texttt{tests}_d(s)$ to denote the test suite produced by this algorithm from a specification s. Next we briefly explain the behaviour of the algorithm. The algorithm keeps track of states of the global test case being built and of the traversal of the specification by storing pairs (Q', s^t, σ) that indicate that the specification could be in any of the states belonging to Q', that the state of the global test case that we are constructing is s^t, and that the sequence of actions that we have performed so far is σ. The algorithm constructs global test cases by iteratively applying one of the following three possibilities. First, conclude a part of the global test case (this possibility can be seen as the *base case*). Second, add an input to the global test case. The algorithm adds a transition labelled by this input, updates the set of auxiliary tuples and considers all possible outputs at the corresponding port. If the output is expected by the specification then testing can continue after receiving the output; otherwise, the algorithm should reach a state where no further input can be provided. The third possibility considers the case where the global test case patiently waits to receive an output; the possibility of receiving no output, represented by δ, has also to be considered.

Algorithm 2
Input: Specification $s = (Q, I, O, T, q_{in})$.
Output: Global test case $t = (Q_t, I, O \cup \{\delta\}, T_t, q^t_{in})$.

{Initialisation}
$S_{aux} := \{(q_{in}$ **after** $\epsilon, q^t_{in}, \epsilon)\}$; $Q_t := \{q^t_{in}\}$; $T_t := \emptyset$;

{Main loop}
while $S_{aux} \neq \emptyset$ do

- Choose $(Q', s^t, \sigma) \in S_{aux}$;
- $S_{aux} := S_{aux} - \{(Q', s^t, \sigma)\}$;
- Choose one of the following three possibilities:
  1. {Base case: This branch of the test is concluded}
     (a) $s^t := \perp$
  2. {Add an input ?i}
     (a) Let $q' \notin Q_t$ be a fresh state; $Q_t := Q_t \cup \{q'\}$;
     (b) $T_t := T_t \cup \{(s^t, ?i, q')\}$;
     (c) $S_{aux} := S_{aux} \cup \{(Q'$ **after** $?i, q', \sigma?i)\}$;
     (d) For all $!o \in O$ such that **fail**$(s, \sigma!o)$ do {These are unexpected outputs: Construct $\perp$ after them}
        i. Let $q' \notin Q_t$ be a fresh state; $Q_t := Q_t \cup \{q'\}$; $q' := \perp$; $T_t := T_t \cup \{(s^t, !o, q')\}$
     (e) For all $!o \in O$ such that $\neg$**fail**$(s, \sigma!o)$ do {These are expected outputs: Testing can continue after them}
        i. Let $q' \notin Q_t$ be a fresh state; $Q_t := Q_t \cup \{q'\}$; $T_t := T_t \cup \{(s^t, !o, q')\}$;
        ii. $S_{aux} := S_{aux} \cup \{(Q'$ **after** $!o, q', \sigma!o)\}$
  3. {Wait for an output}
     (a) For all $!o \in O \cup \{\delta\}$ such that **fail**$(s, \sigma!o)$ do {These are unexpected outputs: Construct $\perp$ after them}
        i. Let $q' \notin Q_t$ be a fresh state; $Q_t := Q_t \cup \{q'\}$; $q' := \perp$; $T_t := T_t \cup \{(s^t, !o, q')\}$;
     (b) For all $!o \in O \cup \{\delta\}$ such that $\neg$**fail**$(s, \sigma!o)$ do {These are expected outputs: Testing can continue after them}
        i. Let $q' \notin Q_t$ be a fresh state; $Q_t := Q_t \cup \{q'\}$; $T_t := T_t \cup \{(s^t, !o, q')\}$;
        ii. $S_{aux} := S_{aux} \cup \{(Q'$ **after** $!o, q', \sigma!o)\}$

output t

Fig. 18 Global test cases generation algorithm.

Now we prove the soundness and completeness of $\texttt{tests}_d(s)$. First, by using Proposition 16, it is obvious that i **dioco** s implies that for every global test case $t \in \texttt{tests}_d(s)$ we have i **passes** t for s. The other implication is stated in the following result (the proof is given in the appendix of the paper).

Proposition 22 *Let s, i be IOTSs. If for every global test case $t \in \texttt{tests}_d(s)$ we have i* **passes** *t for s, then we also have i* **dioco** *s.*

From Proposition 22 we immediately obtain the desired result.

Corollary 2 *Let s, i be IOTSs. i* **dioco** *s if and only if for every global test case $t \in \texttt{tests}_d(s)$, i* **passes** *t for s.*

In order to derive a controllable test suite for s, that we call $\texttt{tests}_c(s)$, it is enough to consider the test suite $\texttt{tests}_d(s)$ and remove uncontrollable test cases by applying the algorithm presented in Section 6.3. Therefore, we obtain the same result (the proof is very similar) as the one given in Proposition 22 for the **dioco** relation.

Proposition 23 *Let s, i be IOTSs. If for every local test case $t \in \texttt{tests}_c(s)$, i* **passes** *t for s, then i* **c-dioco** *s.*

Corollary 3 *Let s, i be IOTSs. Then, i* **c-dioco** *s if and only if for every local test case $t \in \texttt{tests}_c(s)$ we have that i* **passes** *t for s.*

8 Dealing with specifications that are not input-enabled

Our definition of **dioco** assumes that the specification is input-enabled. However, this restriction can lead to us having to produce larger specifications than would otherwise be required and in practice many specifications are not input-enabled. In this section we therefore extend our definition of **dioco** to specifications that are not input enabled. However, as usual we still require implementations to be input-enabled.

The approach used in **ioco** and many other implementation relations is to say that if an input $?i$ is received in a situation in which the behaviour is not specified then the implementation is allowed to behave in any way after this. However, typically there are two alternatives in defining when the response to an input $?i$ after a trace σ is specified:

- there is a sequence of transitions of s that is consistent with the application of $?i$ after σ; or
- after every sequence of transitions of s that is consistent with σ, the response to $?i$ is specified.

In this section we define two corresponding implementation relations. In order to motivate the difference between these relations, let us consider a specification s that can perform the trace $?i_1?i_2?i_3(!o_1, -)$ and then terminate and the global trace $\sigma = ?i_2?i_1?i_3(!o'_1, -)$, with $!o'_1 \neq !o_1$. We might ask whether σ is an acceptable behaviour if s is the specification. There are at least the following ways of looking at this and these lead to different conclusions.

1. No trace of s starts with $?i_2?i_1?i_3$ and so the behaviour after $?i_2?i_1?i_3$ is not specified. As a result, all outputs after $?i_2?i_1?i_3$ are allowed. In particular $?i_2?i_1?i_3(!o'_1, -)$ is allowed. However,

$$?i_2?i_1?i_3(!o'_1, -) \sim ?i_1?i_2?i_3(!o'_1, -)$$

and so $?i_1?i_2?i_3(!o'_1, -)$ must be allowed by the specification. This is despite the specification explicitly including $?i_1?i_2?i_3(!o_1, -)$ and having no trace equivalent to $?i_1?i_2?i_3(!o'_1, -)$ under $\sim$.
2. The specification contains the trace $?i_1?i_2?i_3(!o_1, -)$ and also $?i_1?i_2?i_3(!o_1, -) \sim ?i_2?i_1?i_3(!o_1, -)$ holds. Thus, the specification gives an output $(!o_1, -)$ after $?i_2?i_1?i_3$ but does not give $(!o'_1, -)$ after this. Therefore, the trace σ is not allowed by s.

These two options present different pros and cons. The first one has a potentially non-intuitive property as identified above. Interestingly, the trace σ is allowed by s under **ioco** and so this example shows that the second approach can lead to implementation relations that are stronger than **ioco**.

Both of these approaches decide whether the response to an input $?i$ after some σ is specified by determining whether the response after $?i$ is specified after traces indistinguishable from σ. However, we may not be able to determine when input $?i$ was applied in a trace and so we may not know that it was applied after a trace equivalent (under $\sim$) to a given σ. Let us consider, for example, a specification s in which we have $(!o_1, -)(-, !o_2)?i_3$. The set $\mathcal{T}r^*(s)$ contains a trace that is equivalent to $(!o_1, -)?i_3(-, !o_2)$ under $\sim$ and so we might argue that there is a specified response to $?i_3$ after $(!o_1, -)$. Similarly, there is a response to $?i_3$ specified after $(-, !o_2)$ since $(!o_1, -)(-, !o_2)?i_3 \sim (-, !o_2)?i_3(!o_1, -)$. Thus, we might also consider the sequences that could have preceded $?i$ given the observed local traces and in this section we discuss two implementation relations that do this.

The first implementation relation effectively says that the response to $?i$ is not specified, and so can lead to any output, if it is applied in a trace σ that is equivalent to a trace σ' in which the response to $?i$ is not specified.

Definition 21 Let s, i be `IOTSs` with the same input and output alphabets such that i is input-enabled. We write i **dioco**$_1$ s if for every trace $\sigma \in \mathcal{A}ct^\omega$ of i there exists a trace $\sigma' = a'_1, a'_2, \ldots$ with $\sigma' \sim \sigma$ such that one of the following holds:

1. We have that σ' is a trace of s; or
2. There exists input a'_j such that $a'_1, \ldots, a'_{j-1} \in \mathcal{T}r^*(s)$ and $a'_1, \ldots, a'_j \notin \mathcal{T}r^*(s)$.

The first case simply says that the behaviour σ is acceptable if it is equivalent to a behaviour (global trace) of s under $\sim$. The second case says that σ is acceptable if it is equivalent under $\sim$ to a trace that starts with a prefix $a'_1, \ldots, a'_{j-1}$ of a trace of s followed by an input a'_j such that the response to a'_j after $a'_1, \ldots, a'_{j-1}$ is not specified.

The second implementation relation corresponds to the situation in which we want to say that the response to input $?i$ in a trace σ is specified if σ is equivalent to a trace σ' in which the response to $?i$ is specified.

Definition 22 Let s, i be `IOTSs` with the same input and output alphabets such that i is input-enabled. We write i **dioco**$_2$ s if for every trace σ of i there exists a trace $\sigma' = a'_1, a'_2, \ldots$ with $\sigma' \sim \sigma$ such that one of the following holds:

1. We have that σ' is a trace of s; or
2. There exists $j \geq 1$ with a'_j an input such that the following hold:
   (a) There exists $\sigma_1 \in \mathcal{T}r^\omega(s)$ and $\sigma'_1 \sim \sigma_1$ that has prefix $a'_1, \ldots, a'_{j-1}$; and
   (b) For all $\sigma_1 \in \mathcal{T}r^\omega(s)$ there does not exist $\sigma'_1 \sim \sigma_1$ that has prefix $a'_1, \ldots, a'_j$.

The first part of the second condition says that $a'_1, \ldots, a'_{j-1}$ must be consistent with some trace of s: we must have a trace with prefix $a'_1, \ldots, a'_{j-1}$ that is equivalent to a trace of s under $\sim$. The second part of the second condition requires that $a'_1, \ldots, a'_j$ is not consistent with a trace of s.

The implementation relations **dioco**$_1$ and **dioco**$_2$ differ in when they say that the response to an input $?i$ is not specified (and so can be followed by any output). It might appear that **dioco**$_1$ places stricter requirements on when the response to an input is specified since for the response to $?i$ in σ to not be specified it is sufficient that σ can be rewritten to a trace whose prefix is a trace of s followed by the input of $?i$ in a state where the response to this input is not specified. In such a situation, it might be that σ can also be rewritten to a trace in which the response to $?i$ is specified and so we would expect there to be implementations that pass **dioco**$_1$ but not **dioco**$_2$. However, under **dioco**$_1$ we compare permutations (under $\sim$) of the trace σ of the SUT with traces of s while under **dioco**$_2$ we compare them with permutations (under $\sim$) of traces of s and this leads to situations in which an implementation passes **dioco**$_2$ but not **dioco**$_1$. In the next section we give alternative characterisations of **dioco**$_1$ and **dioco**$_2$ in terms of operators acting on sets of traces. The proof of the following result is given in the appendix of the paper.

Proposition 24 *Let s, i be `IOTSs` with the same input and output alphabets such that i is input-enabled. We might have that i* **dioco**$_2$ *s but also that i* **dioco**$_1$ *s does not hold. In addition, it is possible that i* **dioco**$_1$ *s but i* **dioco**$_2$ *s that does not hold.*

We now show how these implementation relations relate to **ioco**. The proof is again given in the appendix of the paper.

Proposition 25 *Let s, i be IOTSs with the same input and output alphabets such that i is input-enabled. If i* **ioco** *s then we must have that i* **dioco**$_1$ *s. However, it is possible that i* **dioco**$_1$ *s but that i* **ioco** *s does not hold. It is possible that i* **dioco**$_2$ *s but also that i* **ioco** *s does not hold. Similarly, it may be that i* **ioco** *s but that i* **dioco**$_2$ *s does not hold.*

8.1 An alternative characterisation

We have defined two implementation relations for testing against an IOTS that is not input-enabled. We now give alternative characterisations of these implementation relations. These characterisations show that the implementation relations have certain expected properties. We first define some operations on sets of infinite traces.

Definition 23 Let $A \subseteq \mathcal{A}ct^\omega$ be a set of infinite traces. We define the following:

1. pref(A) denotes the set of prefixes of sequences from A and so $A \subseteq$ pref(A).
2. $\mathcal{S}y(A) = \{\sigma' | \exists \sigma \in A : \sigma' \sim \sigma\}$.
3. $\mathcal{C}(A) = A \cup \left\{ \sigma_1 ?i \sigma_1' \;\middle|\; \begin{array}{l} ?i \in I \wedge \sigma_1 \in \text{pref}(A) \wedge \\ \sigma_1 ?i \notin \text{pref}(A) \wedge \sigma_1' \in \mathcal{A}ct^\omega \end{array} \right\}$.

Essentially, $\mathcal{S}y$ takes a set A of traces and produces the set of traces that are equivalent to those in A under $\sim$. The function $\mathcal{C}$ takes a set A of traces that could correspond to those defined by a process and completes this by adding every trace that can be formed by extending a prefix σ_1 of a trace in A by an input $?i$ if no trace in A has prefix $\sigma_1 ?i$.

We now give alternative characterisations of **dioco**$_1$ and **dioco**$_2$ (the proofs are given in the appendix).

Proposition 26 *Let s, i be IOTSs with the same input and output alphabets such that i is input-enabled. Then, i* **dioco**$_1$ *s if and only if for all $\sigma \in \mathcal{T}r^\omega(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{T}r^\omega(s)))$.*

Proposition 27 *Let s, i be IOTSs with the same input and output alphabets such that i is input-enabled. Then, i* **dioco**$_2$ *s if and only if for all $\sigma \in \mathcal{T}r^\omega(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^\omega(s))))$.*

We can now prove that our implementation relations have the expected property that if two traces are indistinguishable (equivalent under $\sim$) then either both are allowed by a specification or neither is allowed. However, before doing this we define what it means for a trace σ to be allowed by a specification under **dioco**$_1$ or **dioco**$_2$. Moreover, based on Definition 21, given a trace $\sigma \in \mathcal{A}ct^\omega$, we can say what it means for σ to be acceptable given specification s.

Definition 24 Let s be an IOTS and $\sigma \in \mathcal{A}ct^\omega$ be a trace. We have that σ **passes**$_1$ for s if one of the following holds:

1. There exists a trace σ' of s such that $\sigma' \sim \sigma$; or
2. There exists a trace $\sigma' = a_1', a_2', \ldots$ such that $\sigma' \sim \sigma$, and input a_j' such that $a_1', \ldots, a_{j-1}' \in \mathcal{T}r^*(s)$ and $a_1', \ldots, a_j' \notin \mathcal{T}r^*(s)$.

We have that σ **passes**$_2$ for s if there exists a trace $\sigma' = a_1', \ldots$ with $\sigma' \sim \sigma$, such that one of the following holds:

1. We have that σ' is a trace of s; or
2. There exists $j \geq 1$ with a_j' an input such that the following hold:
   (a) There exists $\sigma_1 \in \mathcal{T}r^\omega(s)$ and $\sigma_1' \sim \sigma_1$ that has prefix $a_1', \ldots, a_{j-1}'$; and
   (b) For all $\sigma_1 \in \mathcal{T}r^\omega(s)$ there does not exist $\sigma_1' \sim \sigma_1$ that has prefix $a_1', \ldots, a_j'$.

We can now show that under both implementation relations we have that if σ is an acceptable global trace and $\sigma' \sim \sigma$, then σ' is an acceptable global trace. This is a property we would expect implementation relations for distributed testing to have: Since we cannot distinguish between σ and σ' based on the observations we make, either both should pass or both should fail.

Proposition 28 *Let s be an IOTS and $\sigma, \sigma' \in \mathcal{A}ct^\omega$ be traces such that $\sigma \sim \sigma'$. We have that σ* **passes**$_1$ *s if and only if σ'* **passes**$_1$ *s and σ* **passes**$_2$ *s if and only if σ'* **passes**$_2$ *s.*

Proof The proof of the first result follows from Proposition 26 while the proof of the second result follows from Proposition 27.

For **dioco**$_1$ and **dioco**$_2$, the notion of a test case is identical to that used with **dioco**. It is relatively straightforward to adapt Algorithm 2 to produce test cases when the specification is not input-enabled, though the resultant algorithm is more complicated. Specifically, we need to make two main changes. First, we only apply an input if, at that point, it is suitable for the given implementation relation and specification. Second, the function **fail** has to be changed to reflect the implementation relation used.

9 Related Work

Work on distributed testing started in the context of protocol conformance testing and so focussed on testing from a deterministic finite state machine (DFSM). The original work concentrated on controllability problems [49,13] and was followed by research that identified the notion of an observability problem, in which the trace that occurred is not a trace of the specification but is equivalent to a trace of the specification [14]. This work led to the development of methods that generate test sequences that are free of controllability problems and/or observability problems [6,11,27,40,50,57, 56,52,28] when testing from a DFSM. Given a DFSM, it is straightforward to define what it means for a path from the initial state to define a controllable test sequence: we require that each input (after the first) is sent by a tester that observed input and/or output in the previous transition. Thus, a transition t from state s_1 to state s_2 with input $?i$ and output $!o$ can be followed by input at any port p such that the projection of $?i!o$ at p is non-empty. If P is the set of ports $p \in \mathcal{P}$ such that $\pi_p(?i!o) \neq \epsilon$ then we can produce a copy s_2^P of the state s_2 and this is used to represent the situation in which the DFSM has been taken to state s_2 and the next input can be supplied at any port in P without causing a controllability problem. Thus, for each state $s_1^{P'}$ such that $?i$ is input at a port in P', we have a copy of t from $s_1^{P'}$ to s_2^P. Based on this, it is possible to devise a DFSM in which all paths are controllable and, in addition, all controllable paths of the original DFSM are represented. It is then straightforward to generate controllable test sequences (see, for example, [27]). Methods for overcoming observability problems typically instead attempt to include a transition t within paths such that for every port p there is a path in which an incorrect output at p in t must be observed. For example, if t has input $?i$ at p and is followed by another transition t' with input $?i'$ at p in some path then we can identify the output produced at p (in testing) in response to $?i$ since this output is immediately preceded by $?i$ and followed by $?i'$.

It is straightforward to show that there are DFSMs in which it is not possible to achieve a test criterion, such as executing all transitions, without encountering controllability and/or observability problems. There has thus been work on schemes for overcoming controllability and observability problems by allowing the testers to exchange coordination messages through external channels [9,46,53]. The idea is that coordination messages are sent to a tester at a particular port to provide it with additional information that allows it to overcome the controllability and observability problems in a given test sequence. Let us suppose, for example, that in testing we are to follow a transition t with input $?i$ at port p and output $!o$ by an input $?i'$ at a port q such that $\pi_q(?i!o) = \epsilon$. This controllability problem, caused by the tester at q not observing input or output in t, can be overcome by the tester at p sending a coordination message to the tester at q after it supplies input $?i$: the tester at q knows that it can send $?i'$ once it has received this coordination message. In addition, the problem of overcoming controllability problems by allowing the testers to exchange coordination messages has been examined in the context in which the DFSM has timing constraints [38]. Here the latency introduced by the exchange of coordination messages has to be considered when devising test sequences. However, all of this work assumes that the normal implementation relation is used, in which we require that every trace of the SUT is also a trace of the specification.

Recent work has explored the effect, when testing from a DFSM, of restricting testing to test cases that cause no controllability problems [29,21]. Specifically, [29] explored the problem of distinguishing two states of a DFSM when using test cases that cause no controllability problems and showed that such test cases can be produced in polynomial time, where they exist. In contrast, [21] explored the set of finite state machines (FSMs) that cannot be distinguished from a given DFSM M when testing with controllable test cases. It transpires that the set of languages defined by these FSMs forms a bounded lattice. The maximal and minimal elements of this lattice are defined by FSMs, called M_{max} and M_{min} respectively, that can be produced in polynomial time. The FSM M_{min} has the interesting property that for an FSM to be indistinguishable from M in controllable testing it must contain all of the traces of M_{min}. In addition, we have that an FSM M' is indistinguishable from M in controllable testing if and only if every trace of M' is also a trace of M_{max}. The FSM M_{max} thus allows us to explore the consequences of only using controllable test cases when testing from a DFSM M.

While DFSMs are suitable for specifying some important classes of systems, many distributed systems are nondeterministic. In addition, in FSMs input and output alternate and FSMs have finite sets of states. There has therefore been interest in distributed testing from an IOTS. One line of research has defined the implementation relation **mioco** [7]. This relation allows the SUT to not be input-enabled; it allows there to be states in which the SUT will block certain input but if it will block an input $?i_p$ at port p when in state q then it must block all input at p when in state q. The idea is that an SUT might choose not to receive input at

a particular port. While **mioco** is designed for testing from an IOTS that has multiple ports, it assumes that there is a single tester that controls and observes all of the ports; this tester can observe the global behaviour (trace) of the system. As a result, under **mioco** we have that the traces of the SUT are compared to traces of the specification, rather than being compared up to $\sim$.

In addition to **mioco**, there has been work that shows how a global tester can be distributed to form local testers [37]. The method given sends (synchronous) message between the testers whenever an input is sent or an output is received and so effectively establishes a global tester, called an Election Service. This line of work therefore shows how a set of local testers, that can communicate through synchronous messages, can be used to make global observations. However, it relies on many messages being sent for each event and the requirement that the testers can synchronise through message exchange appears to be a restriction. It has been shown that the scalability problems caused by sending many coordination messages can be reduced if the testers are connected in the form of a tree [12].

An SUT might have an input, called a *status message*, that produces an output that identifies the state of the SUT. It has been shown that the presence of a status message can simplify testing when there is a single port [31]. Recent work has shown that the presence of a status message can sometimes be used to overcome controllability problems, in testing from a DFSM, if suitable test sequences are chosen [20]. It might be possible to implement something similar to a status message using monitors, which have been developed for monitoring and debugging distributed systems [41,18,58,3].

An alternative approach, to those discussed above, is to produce a new formalism for modelling distributed systems. One such formalism, that has recently been devised, uses partial orders on inputs and outputs to label transitions [19,4]. This approach has the advantage of being able to, for example, model a situation in which several outputs can be produced but the order in which these are produced is not specified. The implementation relation used compares traces of the SUT and the specification, rather than comparing them up to $\sim$. Thus, this alternative approach can be seen as adapting the type of model/formalism used instead of the implementation relation. It would be interesting to explore the use of implementation relations similar to **dioco** with such models.

All of the work described above compares traces of the SUT with traces of the specification. In contrast our implementation relations model the effect of distributed testing on our ability to make observations regarding the SUT. There has, however, been work that has applied a similar approach for *queued testing*, in which the tester interacts with the SUT through asynchronous first in first out (FIFO) channels [32–34]. Since communications are asynchronous, the input provided by the tester is received after it is sent and the output produced by the SUT is observed after it is sent. As a result, the trace observed by the tester need not be the one produced by the SUT. Thus, this work on queued testing explored the effect of asynchronous FIFO channels on the observational power of the tester.

This paper has defined implementation relations for distributed testing from an IOTS as well as exploring controllability and test generation in this context. It builds on previous work by the authors [24,25], which appears to be the first work to define implementation relations for distributed testing from an IOTS in which we compare traces of the SUT and the specification up to $\sim$. Specifically, [25] defined the implementation relation that we call $\mathbf{dioco}_\delta$ and [24] defined what it meant for a test case to be controllable. However, $\mathbf{dioco}_\delta$ is strictly weaker than our new implementation relation **dioco**. It also does not have some of the desirable properties, such as being stronger than **p-dioco** and being equivalent to **ioco** when testing single-port systems. In addition, the work on controllability allowed only a restricted type of IOTS in which a transition can only produce a single output rather than a tuple of outputs at different ports [24]; the generalisation in this paper has required new definitions and algorithms. These previous papers also only considered IOTSs in which there are two ports and did not consider the important situation in which the specification is not input-enabled. Let us note that the concept of the agent at a port only making local observations has been explored in the context of refinement in CSP but the issues considered were quite different [36].

10 Conclusions

This paper has investigated implementation relations for input output transition systems that have multiple physically distributed interfaces/ports. This situation corresponds to the (standardised) 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 systems, input output transition systems are more general.

When there are distributed ports and a separate agent at each port, each agent can only observe the actions at its port. As a result, it is not possible to reconstruct the global trace that occurred using only

the observations made at each port. This weakness affects the ability of testers to check the conformance of an implementation with respect to a specification. This paper defined three new implementation relations for input-enabled specifications and showed how they relate to one another and to the implementation relation **ioco** traditionally used when there is only one port.

If no agent can receive information regarding observations made at more than one port of the system under test (SUT) then a trace produced by the SUT only distinguishes the SUT from the specification if there is a port at which the traces produced by the SUT differ from all traces of the specification. This situation leads to the implementation relation **p-dioco**. By contrast, if the local observations can be brought together then we obtain a stronger implementation relation **dioco**. The implementation relation **dioco** is stronger than that defined previously [25] since the previous relation, that we call $\mathbf{dioco}_\delta$ in this paper, only considered traces of the SUT that end in quiescence. Importantly, unlike the implementation relation defined in [25], for a single-port SUT i and specification s we have that i **dioco** s if and only if i **ioco** s.

It is normal to require a test case to be deterministic. However, the existence of multiple ports requires an additional property: If the local tester at a port is to decide between alternative actions then the choice must be based on differences in observations at its port. A test case that has this property is said to be controllable and it is possible to decide whether a test case is controllable in polynomial time. The implementation relation **c-dioco** captures the situation in which the use of a system must correspond to a controllable test case.

Having defined three new implementation relations, we proved that **dioco** is strictly weaker than **ioco**, **p-dioco** and **c-dioco** are strictly weaker than **dioco**, and **p-dioco** and **c-dioco** are incomparable. In addition, we have introduced two new implementation relations that deal with non input-enabled specifications and we have compared them with **ioco**. We also provided an alternative characterisation of these implementation relations, showing that they have certain expected properties. Finally, this paper also describes a test generation algorithm that can be used to provide a set of *essential* test cases.

There are several avenues of future work. First, we are working on the parameterisation of the implementation relations with a set of sequences to enforce certain *synchronisation* points. We have taken as preliminary step [26], where synchronisation points are originally added to the specification of the system. 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. An interesting future line is to study how to exploit the use of coordination messages for testing from input output transitions systems. In this line, the existence of a global clock can also help to (partially) synchronise local testers. However, this assumption is sometimes unrealistic. We are working on an approach where each local tester has an individual clock. These clocks can run at different speeds but we assume that the difference between them, in terms of percentage since they were initialised, is not *big*. Therefore, we cannot distinguish the order in which different actions were performed at different ports if the difference between the recorded times is an admissible error due to the different speeds of clocks, but we can decide this order if the actions were performed at very different times. Another line of research consists of defining test objectives for the distributed test architecture. This would allow the tester to overcome the *completeness in the limit* situation that we described in Section 7 since the test derivation algorithm would concentrate on covering the specified test purposes. Finally, recent work has described systems in which an operation is triggered by receiving input at more than one port [19,4]. It would be interesting to extend our work to such systems.

Acknowledgements

We would like to thank Ana Cavalcanti and Marie-Claude Gaudel for the careful reading and their useful comments on a previous version of this paper. We would also like to thank the anonymous referees for their constructive reviews that have helped to further strengthen the paper.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Transactions on Software Engineering **29**(7), 623–633 (2003)
2. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Towards a tool environment for model-based testing with AsmL. In: 3rd Int. Workshop on Formal Approaches to Testing of Software, FATES'03, LNCS 2931, pp. 252–266. Springer (2003)
3. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: 17th Australian Software Engineering Conference, ASWEC'06, pp. 243–252. IEEE Computer Society (2006)
4. Bochmann, G.v., Haar, S., Jard, C., Jourdan, G.V.: Testing systems specified as partial order input/output automata. 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, pp. 169–183. Springer (2008)
5. Bosik, B.S., Uyar, M.Ü.: Finite state machine based formal methods in protocol conformance testing. Computer Networks & ISDN Systems **22**, 7–33 (1991)
6. Boyd, S., Ural, H.: The synchronization problem in protocol testing and its complexity. Information Processing Letters **40**(3), 131–136 (1991)
7. 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, pp. 67–82. Kluwer Academic Publishers (1998)
8. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: 4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067, pp. 187–195. Springer (2001)
9. Cacciari, L., Rafiq, O.: Controllability and observability in distributed testing. Information and Software Technology **41**(11–12), 767–780 (1999)
10. Chen, J., Hierons, R.M., Ural, H.: Testing in the distributed test architecture. In: Formal Methods and Testing, LNCS 4949, pp. 157–183. Springer (2008)
11. Chen, W., Ural, H.: Synchronizable checking sequences based on multiple UIO sequences. IEEE/ACM Transactions on Networking **3**, 152–157 (1995)
12. Cunha de Almeida, E., Marynowski, J., Sunyé, G., Traon, Y.L., Valduriez, P.: Efficient distributed test architecture for large-scale systems. In: 22nd Int. Conf. on Testing Software and Systems, ICTSS'10, LNCS 6435, pp. 174–187. Springer (2010)
13. Dssouli, R., Bochmann, G.v.: Error detection with multiple observers. In: 5th WG6.1 Int. Conf. on Protocol Specification, Testing and Verification, PSTV'85, pp. 483–494. North-Holland (1985)
14. Dssouli, R., Bochmann, G.v.: Conformance testing with multiple observers. In: 6th WG6.1 Int. Conf. on Protocol Specification, Testing and Verification, PSTV'86, pp. 217–229. North-Holland (1986)
15. En-Nouaary, A., Dssouli, R., Khendek, F.: Timed Wp-method: Testing real time systems. IEEE Transactions on Software Engineering **28**(11), 1024–1039 (2002)
16. Farchi, E., Hartman, A., Pinter, S.: Using a model-based test generator to test for standard conformance. IBM Systems Journal **41**(1), 89–110 (2002)
17. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ACM SIGSOFT Symposium on Software Testing and Analysis, ISSTA'02, pp. 112–122. ACM Press (2002)
18. Gunter, D., Tierney, B., Crowley, B., Holding, M., Lee, J.: Netlogger: A toolkit for distributed system performance analysis. In: 8th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS'00, pp. 267–273. IEEE Computer Society (2000)
19. 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, pp. 171–185. Springer (2007)
20. Hierons, R.M.: Using status messages in the distributed test architecture. Information & Software Technology **51**(7), 1123–1130 (2009)
21. Hierons, R.M.: Canonical finite state machines for distributed systems. Theoretical Computer Science **411**(2), 566–580 (2010)
22. Hierons, R.M., Bogdanov, K., Bowen, J., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A., Vilkomir, S., Woodward, M., Zedan, H.: Using formal methods to support testing. ACM Computing Surveys **41**(2) (2009)
23. Hierons, R.M., Bowen, J., Harman, M. (eds.): Formal Methods and Testing, LNCS 4949. Springer (2008)
24. Hierons, R.M., Merayo, M.G., Núñez, M.: Controllable test cases for the distributed test architecture. In: 6th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'08, LNCS 5311, pp. 201–215. Springer (2008)
25. Hierons, R.M., Merayo, M.G., Núñez, M.: 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, pp. 200–215. Springer (2008)
26. Hierons, R.M., Merayo, M.G., Núñez, M.: Scenarios-based testing of systems with distributed ports. Software - Practice and Experience **41**(10), 999–1026 (2011)
27. Hierons, R.M., Ural, H.: Synchronized checking sequences based on UIO sequences. Information and Software Technology **45**(12), 793–803 (2003)
28. Hierons, R.M., Ural, H.: Checking sequences for distributed test architectures. Distributed Computing **21**(3), 223–238 (2008)
29. Hierons, R.M., Ural, H.: The effect of the distributed test architecture on the power of testing. The Computer Journal **51**(4), 497–510 (2008)
30. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley (2006)
31. Huang, C.M., Chang, Y.I., Liu, M.: A computer-aided incremental protocol test sequence generation: the production system approach. In: IEEE Annual Phoenix Conference on Computers and Communications, pp. 608–614. IEEE Computer Society (1991)
32. Huo, J., Petrenko, A.: On testing partially specified IOTS through lossless queues. In: 16th Int. Conf. on Testing Communicating Systems, TestCom'04, LNCS 2978, pp. 76–94. Springer (2004)
33. Huo, J., Petrenko, A.: Covering transitions of concurrent systems through queues. In: 16th Int. Symposium on Software Reliability Engineering, ISSRE'05, pp. 335–345. IEEE Computer Society (2005)
34. Huo, J., Petrenko, A.: Transition covering tests for systems with queues. Software Testing, Verification and Reliability **19**(1), 55–83 (2009)
35. 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)
36. Jacob, J.: Refinement of shared systems. In: J. McDermid (ed.) The Theory and Practice of Refinement: Approaches to the Formal Development of Large-Scale Software Systems, pp. 27–36. Butterworths (1989)
37. Jard, C., Jéron, T., Kahlouche, H., Viho, C.: Towards automatic distribution of testers for distributed conformance testing. In: TC6 WG6.1 Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE'98, pp. 353–368. Kluwer Academic Publishers (1998)
38. Khoumsi, A.: A temporal approach for testing distributed systems. IEEE Transactions on Software Engineering **28**(11), 1085–1103 (2002)

39. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE **84**(8), 1090–1123 (1996)
40. 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, pp. 139–153. North-Holland (1993)
41. Mansorui-Samani, M., Sloman, M.: Monitoring distributed systems. IEEE Network **7**(6), 20–30 (1993)
42. Merayo, M.G., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer Networks **52**(2), 432–460 (2008)
43. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: 4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067, pp. 196–205. Springer (2001)
44. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Transactions on Software Engineering **30**(1), 29–42 (2004)
45. Petrenko, A., Yevtushenko, N.: Testing from partial deterministic FSM specifications. IEEE Transactions on Computers **54**(9), 1154–1165 (2005)
46. Rafiq, O., Cacciari, L.: Coordination algorithm for distributed testing. The Journal of Supercomputing **24**(2), 203–211 (2003)
47. Rodríguez, I.: A general testability theory. In: 20th Int. Conf. on Concurrency Theory, CONCUR'09, LNCS 5710, pp. 572–586. Springer (2009)
48. Rodríguez, I., Merayo, M.G., Núñez, M.: $\mathcal{HOTL}$: Hypotheses and observations testing logic. Journal of Logic and Algebraic Programming **74**(2), 57–93 (2008)
49. Sarikaya, B., Bochmann, G.v.: Synchronization and specification issues in protocol testing. IEEE Transactions on Communications **32**, 389–395 (1984)
50. Tai, K.C., Young, Y.C.: Synchronizable test sequences of finite state machines. Computer Networks and ISDN Systems **30**(12), 1111–1134 (1998)
51. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software – Concepts and Tools **17**(3), 103–120 (1996)
52. Ural, H., Whittier, D.: Distributed testing without encountering controllability and observability problems. Information Processing Letters **88**(3), 133–141 (2003)
53. Ural, H., Williams, C.: Constructing checking sequences for distributed testing. Formal Aspects of Computing **18**(1), 84–101 (2006)
54. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann (2007)
55. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: Formal Methods and Testing, LNCS 4949, pp. 39–76. Springer (2008)
56. Wu, W.J., Chen, W.H., Tang, C.: Synchronizable test sequence for multi-party protocol conformance testing. Computer Communications **21**(13), 1177–1183 (1998)
57. Young, Y., Tai, K.: Observational inaccuracy in conformance testing with multiple testers. In: IEEE 1st Workshop on Application-specific Software Engineering and Technology, pp. 80–85. IEEE Computer Society (1998)
58. Zulkernine, M., Seviora, R.: A compositional approach to monitoring distributed systems. In: 3rd Int. Conf. on Dependable Systems and Networks, DSN'02, pp. 763–772. IEEE Computer Society (2002)

Appendix: Additional material

In this appendix we provide the construction of local test cases from global test cases that we mentioned in Section 6.2 and the proofs of some results.

Next we give a method to produce local testers by taking projections of a controllable global test case t. A naïve approach consists in using $t\rceil_p$ for each port p.[2] However, this method produces local tests that contain occurrences of the τ action and/or are nondeterministic. In order to produce *well-formed* local testers, a *determinism* function must be applied to these projections. Since we are interested in the traces that can be executed by a local tester[3], we will denote by $det(t\rceil_p)$ a deterministic automaton having exactly the same traces as $t\rceil_p$. In order to implement the *det* function, we can use the subset construction from automata theory [30]. In general the process of converting a nondeterministic automaton into a deterministic automaton can lead to a combinatorial explosion. However, since test cases are essentially finite trees (instead of general graphs) we can use a specific algorithm that controls the growth in the number of states of local testers. The process of turning such an *intermediate* local tester into a deterministic automaton can proceed through the repeated application of two steps:

1. Merge states q_1 and q_2 if q_1 is the parent of q_2 and $q_1 \xrightarrow{\tau} q_2$.
2. Merge states q_1 and q_2 if there is a state q and event $a \in I_p \cup O_p \cup \{\delta\}$ such that $q \xrightarrow{a} q_1$ and $q \xrightarrow{a} q_2$.

The number of applications of these merge operations is bounded by the number of states of the intermediate local tester (that is equal to the number of states of the original global test case). The reason for having this bound is that each merge operation reduces the number of states. The conversion of a local tester t_p into a deterministic automaton does not suffer a combinatorial explosion, because the number of paths in t_p is bounded by the number of nodes in t_p. The number of prefixes of the regular language $L(t_p)$ is bounded by the number of paths in t_p and the deterministic automaton t'_p of t_p contains at most one state per prefix of $L(t_p)$.

Next we give the proofs of some selected results.

Proposition 22 *Let s, i be* `IOTSs`*. If for every global test case $t \in \mathtt{tests}_d(s)$ we have i* **passes** *t for s, then we also have i* **dioco** *s.*

[2] Let us note that if t is the null process $\perp$, which cannot send any input, then $t\rceil_p = \perp_p$.

[3] We are only interested in the traces defined by a local tester since each tester only observes a local trace.

Proof We prove the result by contradiction: We assume that i **dioco** s does not hold and prove that there exists a global test case $t \in \texttt{tests}_d(s)$ such that i **fails** t for s.

Since i **dioco** s does not hold, there exists $\sigma \in \mathcal{R}(i)$ such that there is not $\sigma' \in \mathcal{R}(s)$ with $\sigma \sim \sigma'$. Similar to the proof of Proposition 16, let σ_1 be the shortest prefix of σ that contains all of the inputs in σ. Trivially this sequence is finite. We distinguish two cases: Either $\textbf{fail}(s, \sigma_1)$ holds or it does not.

If $\textbf{fail}(s, \sigma_1)$ does not hold then there are two more cases. First, if $\sigma_1 = \epsilon$ then $t_1 = \perp$. Second, if $\sigma_1 = a_1 \dots a_n$, with $n \geq 1$, then we construct the global test case $t_1 = (Q, I, O \cup \{\delta\}, T, q_{in})$ by using the following algorithm:

$Q := \{q_{in}\}$; $S_{aux} := \{(q_{in}$ **after** $\epsilon, q_{in}, \epsilon)\}$; $T := \emptyset$;
$\widetilde{\sigma} := \epsilon$; {`Used to store the traversed sequence.`}
{$\widetilde{\sigma}$ `is the` σ `in Algorithm 2.`}
$aux := (q_{in}$ **after** $\epsilon, q_{in}, \epsilon)$;
for $j := 1$ to n do
  Consider aux {aux `belongs to` S_{aux}};
  If $a_j \in I$ then Apply case 2 of Algorithm 2;
    $\widetilde{\sigma} := \widetilde{\sigma} a_j$;
    $aux := (Q_{aux}$ **after** $a_j, q', \widetilde{\sigma})$;
  If $a_j \in O \cup \{\delta\}$ then Apply case 3 of Algorithm 2;
    $\widetilde{\sigma} := \widetilde{\sigma} a_j$;
    $aux := (Q_{aux}$ **after** $a_j, q', \widetilde{\sigma})$;
{`Apply the base case as long as needed`}
For all non-initialised s' where there exists Q' and σ such that $(Q', s', \sigma) \in S_{aux}$ do $s' := \perp$;

Clearly, we have $t_1 \in \texttt{tests}_d(s)$. In particular, $\perp$ always belongs to $\texttt{tests}_d(s)$ since it can be constructed by initially applying the first step. Therefore, i **fails** t_1 for s since both t_1 and i can perform σ, so that $\sigma \in \mathcal{T}r(i, t_1)$, but there does not exist $\sigma' \in \mathcal{R}(s)$ with $\sigma \sim \sigma'$.

If $\textbf{fail}(s, \sigma_1)$ then let σ_2 be the longest prefix of σ_1 such that $\textbf{fail}(s, \sigma_2)$ does not hold but $\sigma_2 a$ is a prefix of σ_1 and $\textbf{fail}(s, \sigma_2 a)$. The sequence σ_2 is obviously shorter than σ_1 and, as such, σ_2 is finite. In addition, a cannot be an input since s is input-enabled and, therefore, if s can perform σ_2, up to $\sim$, and a were an input then s could also perform σ_2 followed by a. We construct t_2 by following the algorithm given for t_1 before, but considering the sequence σ_2, and again we obtain a global test case $t_2 \in \texttt{tests}_d(s)$ such that i **fails** t_2 for s.

Proposition 24 *Let s, i be `IOTSs` with the same input and output alphabets such that i is input-enabled. We might have that i* $\textbf{dioco}_2$ *s but also that i* $\textbf{dioco}_1$ *s does not hold. In addition, it is possible that i* $\textbf{dioco}_1$ *s but i* $\textbf{dioco}_2$ *s that does not hold.*

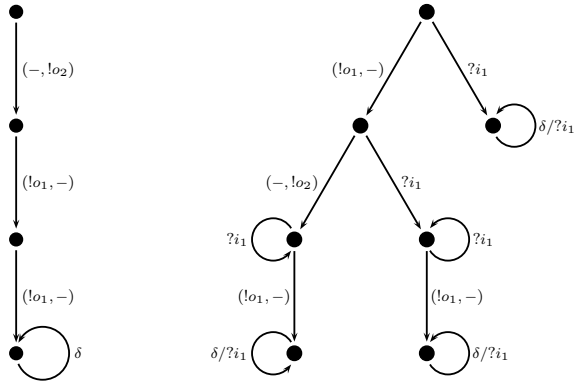


Fig. 19 Processes s and r for Proposition 24.

Proof For the first part, let us consider the processes s, left-hand side of Figure 19, and i, right-hand side of Figure 19. Here $?i_1$ is input at port 1 and is the only allowed input and $(!o_1, -)$ and $(-, !o_2)$ are outputs at ports 1 and 2 respectively. First, we show that i $\textbf{dioco}_2$ s and note that $(!o_1, -)(-, !o_2, -)(!o_1, -)\delta^\omega$ is equivalent to a trace of s under $\sim$. Clearly, all traces that start with $?i_1$ are allowed by s under $\textbf{dioco}_2$ since no trace of s is equivalent under $\sim$ to a trace that starts with $?i_1$. Now let us consider traces of i that start with $(!o_1, -)?i_1$. We can choose $\sigma_1 = (-, !o_2)(!o_1, -)(!o_1, -)\delta^\omega$ and $\sigma_1' = (!o_1, -)(-, !o_2)(!o_1, -)\delta^\omega$ in the first part of the second condition of Definition 22. Importantly, σ_1' starts with $(!o_1, -)$. There is no trace of s that is equivalent to a trace with prefix $(!o_1, -)?i_1$ and so all traces that are equivalent to traces that start with $(!o_1, -)?i_1$ are allowed by s under $\textbf{dioco}_2$ as required.

Now let us consider the relation $\textbf{dioco}_1$ and $\sigma = (!o_1, -)?i_1(!o_1, -)\delta^\omega$. For $\textbf{dioco}_1$ to hold there must be $\sigma' \sim \sigma$ that has a prefix σ_1' that is a prefix of a trace of s and this is followed by an input a such that $\sigma_1' a$ is not a prefix of a trace of s. However, σ is the only trace that is equivalent to σ under $\sim$ and the property clearly does not hold for $\sigma' = \sigma$ since no trace of s starts with $(!o_1, -)$. Thus, we do not have that i $\textbf{dioco}_1$ s as required.

For the second part, we consider specification s that can do $?i_1?i_2(!o_1, -)$ and a process i that can do either $?i_1?i_2(!o_1, -)$ or $?i_2?i_1(!o_1', -)$. Let $\sigma = ?i_2?i_1(!o_1', -)\delta^\omega$. We have that i $\textbf{dioco}_1$ s since this satisfies the second condition of Definition 21. However, since $?i_2?i_1 \sim ?i_1?i_2$ we do not have that σ satisfies the condition of Definition 22 as required. The result thus follows.

Proposition 25 *Let s, i be `IOTSs` with the same input and output alphabets such that i is input-enabled. If i* $\textbf{ioco}$ *s then we must have that i* $\textbf{dioco}_1$ *s. However,*

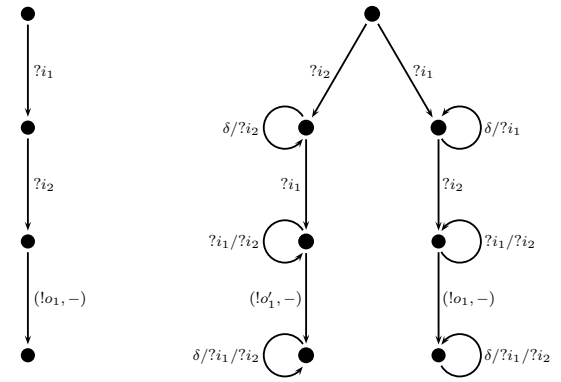


Fig. 20 Processes s and r for Proposition 25.

it is possible that i **dioco**$_1$ s *but that* i **ioco** s *does not hold. It is possible that* i **dioco**$_2$ s *but also that* i **ioco** s *does not hold. Similarly, it may be that* i **ioco** s *but that* i **dioco**$_2$ s *does not hold.*

Proof First, let us assume that i **ioco** s and we are required to prove that i **dioco**$_1$ s. Let $\sigma \in \mathcal{T}r^{\omega}(i)$ be a trace of i and it is sufficient to prove that it must satisfy one of the conditions of Definition 21. If $\sigma \in \mathcal{T}r^{\omega}(s)$ then the result holds immediately and so we assume that $\sigma \notin \mathcal{T}r^{\omega}(s)$. Let $\sigma_1 a$ be a shortest prefix of σ such that $\sigma_1 a \notin \mathcal{T}r^{*}(s)$. Since i **ioco** s we must have that a is an input and by the minimality of σ_1 we have that $\sigma_1 \in \mathcal{T}r^{*}(s)$. Thus, there exists $\sigma_1 \in \mathcal{T}r^{*}(s)$ such that $\sigma_1 a \notin \mathcal{T}r^{*}(s)$ for an input a with $\sigma_1 a$ a prefix of σ. Thus, σ satisfies the second condition of **dioco**$_1$ as required.

For the second part, let us consider a specification s that can do $(!o_1, -)(-, !o_2)$ and then terminates and a process i that can do $(-, !o_2)(!o_1, -)$ and then terminates. Clearly we do not have that i **ioco** s. However, $(!o_1, -)(-, !o_2) \sim (-, !o_2)(!o_1, -)$ and so we have that i **dioco**$_1$ s as required.

For the third part, it is again sufficient to consider a specification s that can do $(!o_1, -)(-, !o_2)$ and then terminates and a process i that can do $(-, !o_2)(!o_1, -)$ and then terminates.

For the fourth part it is sufficient to consider the specification s and the process i depicted in Figure 20, in which $!o'_1 \neq !o_1$. First, since the behaviour after $?i_2$ is not specified in s, we have that i **ioco** s. Now let $\sigma = ?i_2?i_1(!o'_1, -)\delta^{\omega} \in \mathcal{T}r^{\omega}(i)$ and let us consider Definition 22. Clearly there does not exist $\sigma' \sim \sigma$ such that $\sigma \in \mathcal{T}r^{\omega}(s)$. In addition, since $?i_1?i_2 \in \mathcal{T}r^{*}(s)$ and $(o'_1, -)$ is not an output produced by s, the second condition in Definition 22 cannot hold. Thus, we do not have that i **dioco**$_2$ s as required.

Proposition 26 *Let* s, i *be* `IOTSs` *with the same input and output alphabets such that* i *is input-enabled. Then,* i **dioco**$_1$ s *if and only if for all* $\sigma \in \mathcal{T}r^{\omega}(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{T}r^{\omega}(s)))$.

Proof First let us assume that i **dioco**$_1$ s and so we are required to prove that for all $\sigma \in \mathcal{T}r^{\omega}(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{T}r^{\omega}(s)))$. Let $\sigma \in \mathcal{T}r^{\omega}(i)$. Since i **dioco**$_1$ s, there are two cases.

Case 1: There exists a trace σ' of s such that $\sigma' \sim \sigma$ and so $\sigma \in \mathcal{S}y(\mathcal{T}r^{\omega}(s))$. However, for every set $X \subseteq \mathcal{A}ct^{\omega}$, $X \subseteq \mathcal{C}(X)$ and so since $\mathcal{T}r^{\omega}(s) \subseteq \mathcal{C}(\mathcal{T}r^{\omega}(s))$, $\mathcal{S}y(\mathcal{T}r^{\omega}(s)) \subseteq \mathcal{S}y(\mathcal{C}(\mathcal{T}r^{\omega}(s)))$. Thus, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{T}r^{\omega}(s)))$ holds, as required, since $\sigma \in \mathcal{S}y(\mathcal{T}r^{\omega}(s))$.

Case 2: There exists $\sigma' = a'_1, a'_2, \ldots$ such that $\sigma' \sim \sigma$ and input a'_j such that $a'_1, \ldots a'_{j-1} \in \mathcal{T}r^{*}(s)$ and $a'_1, \ldots, a'_j \notin \mathcal{T}r^{*}(s)$. By definition, this implies that for all $\sigma_2 \in \mathcal{A}ct^{\omega} \cup \mathcal{A}ct^{*}\{\delta^{\omega}\}$, $a'_1, \ldots, a'_j\sigma_2 \in \mathcal{C}(\mathcal{T}r^{\omega}(s))$. Thus, $\sigma' \in \mathcal{C}(\mathcal{T}r^{\omega}(s))$ and so $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{T}r^{\omega}(s)))$ as required.

Now let us assume that for all $\sigma \in \mathcal{T}r^{\omega}(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{T}r^{\omega}(s)))$ and so we are required to prove that i **dioco**$_1$ s. Let $\sigma \in \mathcal{T}r^{\omega}(i)$. It is sufficient to prove that σ must satisfy one of the conditions in Definition 21. By the definition of the function $\mathcal{S}y$, there exists some $\sigma' \sim \sigma$ such that $\sigma' \in \mathcal{C}(\mathcal{T}r^{\omega}(s))$. Thus, $\sigma' \in \mathcal{T}r^{\omega}(s) \cup \{\sigma_1 ?i \sigma'_1 | ?i \in I \wedge \sigma_1 \in \mathcal{T}r^{*}(s) \wedge \sigma_1 ?i \notin \mathcal{T}r^{*}(s) \wedge \sigma'_1 \in \mathcal{A}ct^{\omega}\}$. But, this means that the second condition of Definition 21 must hold.

Proposition 27 *Let* s, i *be* `IOTSs` *with the same input and output alphabets such that* i *is input-enabled. Then,* i **dioco**$_2$ s *if and only if for all* $\sigma \in \mathcal{T}r^{\omega}(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$.

Proof First let us assume that i **dioco**$_2$ s. It is necessary to prove that for all $\sigma \in \mathcal{T}r^{\omega}(i)$, we have that $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$. Let σ be some element of $\mathcal{T}r^{\omega}(i)$. There are two cases to consider.

Case 1: There exists a trace σ' of s such that $\sigma' \sim \sigma$ and so $\sigma \in \mathcal{S}y(\mathcal{T}r^{\omega}(s))$. However, for every set $X \subseteq \mathcal{A}ct^{\omega}$, $X \subseteq \mathcal{C}(X)$ and $X \subseteq \mathcal{S}y(X)$. So, $\mathcal{S}y(\mathcal{T}r^{\omega}(s)) \subseteq \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$. As a result, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$ as required.

Case 2: There exists $\sigma' = a'_1, \ldots$, with $\sigma' \sim \sigma$, and $j \geq 1$, with a'_j an input, such that the following hold:

1. There exists $\sigma_1 \in \mathcal{T}r^{\omega}(s)$ and $\sigma'_1 \sim \sigma_1$ that has prefix $a'_1, \ldots, a'_{j-1}$; and
2. for all $\sigma_1 \in \mathcal{T}r^{\omega}(s)$ there does not exist $\sigma'_1 \sim \sigma_1$ that has prefix $a'_1, \ldots, a'_j$.

The first condition implies that $a'_1, \ldots, a'_{j-1}$ is a prefix of a sequence in $\mathcal{S}y(\mathcal{T}r^{\omega}(s))$ and the second condition implies that $a'_1, \ldots, a'_j$ is not a prefix of a sequence in $\mathcal{S}y(\mathcal{T}r^{\omega}(s))$. Thus, $\sigma' \in \mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$ and so $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$ as required.

Now let us assume that for all $\sigma \in \mathcal{T}r^{\omega}(i)$, $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$. Therefore, it is required to prove that i **dioco**$_2$ s. Let $\sigma \in \mathcal{T}r^{\omega}(i)$ and it is sufficient to show that σ must satisfy one of the conditions in Definition 22. Since $\sigma \in \mathcal{S}y(\mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s))))$ there must exist some $\sigma' = a'_1, a'_2, \ldots$, with $\sigma' \sim \sigma$, such that $\sigma' \in \mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$. Clearly, if $\sigma' \in \mathcal{S}y(\mathcal{T}r^{\omega}(s))$ then the first condition of Definition 22 holds and so we assume that $\sigma' \in \mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$ and $\sigma' \notin \mathcal{S}y(\mathcal{T}r^{\omega}(s))$.

By the definition of $\mathcal{C}$, since $\sigma' \in \mathcal{C}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$ and $\sigma' \notin \mathcal{S}y(\mathcal{T}r^{\omega}(s))$, there exists an input a_j such that $a'_1, \ldots, a'_{j-1} \in \text{pref}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$ and $a'_1, \ldots, a'_j \notin \text{pref}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$.

Since $a'_1, \ldots, a'_{j-1} \in \text{pref}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$, there exists $\sigma_1 \in \mathcal{T}r^{\omega}(s)$ and $\sigma'_1 \sim \sigma_1$ such that $a'_1, \ldots, a'_{j-1} \in \text{pref}(\sigma'_1)$ and so the first part of condition 2 of Definition 22 must hold. Furthermore, since $a'_1, \ldots, a'_j \notin \text{pref}(\mathcal{S}y(\mathcal{T}r^{\omega}(s)))$ there does not exist $\sigma_1 \in \mathcal{T}r^{\omega}(s)$ and $\sigma'_1 \sim \sigma_1$ such that $a'_1, \ldots, a'_j \in \text{pref}(\sigma'_1)$ and so the second part of condition 2 of Definition 22 must hold. Thus, σ satisfies the second condition of Definition 22 as required.