A more precise implementation relation for distributed testing

Robert M. Hierons

Department of Computer Science, Brunel University London, Uxbridge, Middlesex, UK
Email: rob.hierons@brunel.ac.uk

There has been significant interest in distributed testing from an input output transition system. Previous work introduced an implementation relation dioco that was defined in terms of an equivalence relation on traces (sequences of observations). This paper considers an alternative approach in which an observation made in testing is a tuple of local traces, one for each tester. This paper defines such an implementation relation dioco$_o$ in terms of the possible observations regarding the system under test and the specification. It shows that dioco$_o$ is strictly weaker than dioco but is equivalent to dioco if processes cannot be output-divergent. Interestingly, this shows that the previous definition of dioco is too strong for output-divergent processes. We also prove that the Oracle problem is NP-complete but can be solved in polynomial time if there is an upper bound on the number of local testers.

Keywords: Distributed Testing; Model-based Testing; Software Testing

Received 00 January 2009; revised 00 Month 2009

1. INTRODUCTION

Software testing is an important form of verification and validation used in software development but is often manual, expensive, and error prone. This has led to the development of methods that allow parts of testing to be automated, with there being significant interest in model based testing (MBT). In MBT, automation is based on a model, with the model often being state-based (see, for example, [1, 2]). There has been particular interest in testing from a model in the form of a finite state machine (FSM) [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] or an input output transition system (IOTS) [14, 15, 16, 17, 18]; while the developer might produce a model in some other formalism, MBT tools often analyse an FSM or IOTS that represents the semantics of the model. This paper concerns testing from an IOTS model.

Most MBT work assumes that a single tester interacts synchronously with the system under test (SUT). However, in some situations the SUT has multiple physically distributed interfaces at which it interacts with its environment and testing involves placing a separate tester at each interface. If the testers cannot synchronise their actions then we are testing in the ISO standardised distributed test architecture [19]. In such situations a local tester will only observe the actions at its interface (port) and so the overall global sequence of inputs and outputs is not observed. This reduces the ability of testing to distinguish between the SUT and specification and so requires the definition of implementation relations (notions of correctness) that reflect the nature of testing. It is important to use a suitable implementation relation since otherwise testing might be unsound (it might declare a correct SUT to be faulty) and might also be inefficient (we might use test cases that cannot lead to faults being found). The implementation relation **dioco** has been defined for distributed testing from an IOTS, with this initially requiring that processes are not output-divergent[1] [20], and then being generalised to allow output-divergent processes [16]. The initial definition compared the quiescent traces[2] of the SUT with the quiescent traces of the specification using an equivalence relation $\sim$. If a process is output-divergent then it may have a trace σ that is not a prefix of a quiescent trace: the initial definition of **dioco** then has the problem that it would effectively ignore σ. The generalisation [16] instead compared a class of infinite traces of the SUT with infinite traces of the specification.

The implementation relation **dioco** is defined in terms of how the global traces of the SUT and the specification relate: **dioco** requires that certain global traces of the SUT are observationally equivalent to global traces of the specification. However, the notion of observation in distributed testing is not a global trace and instead is a tuple $(\sigma_1, \ldots, \sigma_n)$ of local traces: one for each local tester. In testing, the problem of

[1] A process is output-divergent if it has a state from which there is an infinite path that contains no inputs.

[2] A trace σ of a process p is a quiescent trace if it can take p to a state from which the only possible transitions involve input.

checking an observation against a specification (the Oracle problem) is thus that of determining whether an observation $(\sigma_1, \ldots, \sigma_n)$ is consistent with the specification s. Note that another piece of work independently considered the case where an observation is a tuple of local traces, but for situations in which the SUT consists of a set of components and we can observe messages sent between these components [21].

In this paper we define a *verdict function* (that acts as an oracle) that maps an observation to a verdict (pass or fail). We then prove that for processes that are not output-divergent, the power of the verdict function exactly corresponds to **dioco**: a process r conforms to a process s under **dioco** if and only if all observations that can be made of r are allowed for s using the verdict function. This shows that the oracle is suitable for processes that are not output-divergent. Interestingly, it transpires that this property does not hold for the more general setting in which processes can be output-divergent. In particular, it is possible to construct processes r and s such that r does not conform to s under **dioco** but where no (finite) observation can show this. Since testing always makes finite observations, this suggests that the more general definition of **dioco** is too strong for testing and instead it may be better to use an alternative definition, that we call $\mathbf{dioco}_o$, which is defined in terms of finite observations. However, we prove that **dioco** and $\mathbf{dioco}_o$ coincide for processes that are finitely-branching[3] and do not allow starvation: where every infinite global trace, that contains only finitely many inputs, has infinitely many events at all ports.

The paper is structured as follows. Section 2 provides required definitions and Section 3 then defines the notion of an observation and defines a corresponding implementation relation. Section 4 then defines a verdict function for the case where an observation is a tuple of local traces and proves that this is consistent with **dioco** for processes that are not output-divergent. It also defines the new implementation relation $\mathbf{dioco}_o$. Section 5 shows that the problem of computing a verdict is NP-complete but can be solved in polynomial time if there is an upper bound on the number of ports. Section 6 then describes related work. Finally, Section 7 draws conclusions.

2. PRELIMINARIES

Throughout this paper we assume that the SUT has m ports, with the set of names of ports being denoted $\mathcal{P} = \{1, \ldots, n\}$. We assume that the (countable) input alphabet I is partitioned into sets $I_1, \ldots, I_m$ such that for all $p \in \mathcal{P}$ the SUT can receive inputs in I_p at p. Similarly, the (countable) output alphabet O will be partitioned into $O_1, \ldots, O_m$. We assume that the sets $I_1, \ldots, I_m, O_1, \ldots, O_m$ are pairwise disjoint.

[3]A process s is finitely branching if for every state q of s we have that only finitely many transitions leaving q.

Given a set A we will let A^* denote the set of finite sequences of elements of A and A^ω be the set of infinite sequences of elements of A. We will be interested in infinite sequences since the generalised version of the implementation relation **dioco** is defined in terms of infinite traces. Given a sequence σ we will let $pref(\sigma)$ denote the set of prefixes of σ. A sequence is a prefix of itself and the set of prefixes of an infinite sequences include both finite and infinite sequences.

In this paper we assume that any specification or SUT can be represented as an input output transition system.

Definition 2.1. *An input output transition system (IOTS) s is defined by a tuple (Q, I, O, T, q_0) in which Q is a countable set of states, $q_0 \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$, where τ represents internal (unobservable) actions, is the transition relation. A transition $(q, a, q') \in T$ should be interpreted as meaning that from state q it is possible to move to state q' with action $a \in I \cup O \cup \{\tau\}$. We assume that I and O are disjoint and $\tau \notin I \cup O$. State $q \in Q$ is said to be* quiescent *if from q it is not possible to change state or produce output without first receiving input and quiescence is represented by δ. We can extend T, the transition relation, to T_δ by adding the transition (q, δ, q) for each quiescent state q. We assume that quiescence can be observed and so the set of observations is $\mathcal{A}ct = I \cup O \cup \{\delta\}$. Given port $p \in \mathcal{P}$, the set of observations that can be made at p is $\mathcal{A}ct_p = I_p \cup O_p \cup \{\delta\}$.*

Figure 1 gives a previously defined IOTS M_0 that represents a distributed majority voting system [16]. The model interacts with two agents that we call U and L. We included the initial state s_0 twice to simplify the diagram. Initially the system should send $!r_U$ to U and then $!r_L$ to L, with these telling the agents that a poll is to start. Each agent then sends a message to the system with this message giving its vote: $?l_0$ and $?l_1$ denote agent L voting 0 and 1 respectively and $?u_0$ and $?u_1$ denote U voting 0 and 1 respectively. If the system receives identical votes then it sends confirmation to the agents and this is either $!0_L$ and $!0_U$ (to L and U respectively) if the vote was 0 and otherwise $!1_L$ and $!1_U$. If the votes differ then the process is repeated. Where a state s has no transition with an input $?i$ there is an implicit self-loop transition from s to s with input $?i$.

IOTS s is *input-enabled* if for all $q \in Q$ and $?i \in I$ there exists $q' \in Q$ such that $(q, ?i, q') \in T$. As usual, we assume that any process that models the SUT is input-enabled. Thus, the SUT does not block input and it is normal to similarly assume that the environment does not block output produced by the SUT. If an IOTS s is not input-enabled then we say that it is *partial.* We will largely focus on the case where the specification is input-enabled but in Section 4 we briefly discuss the

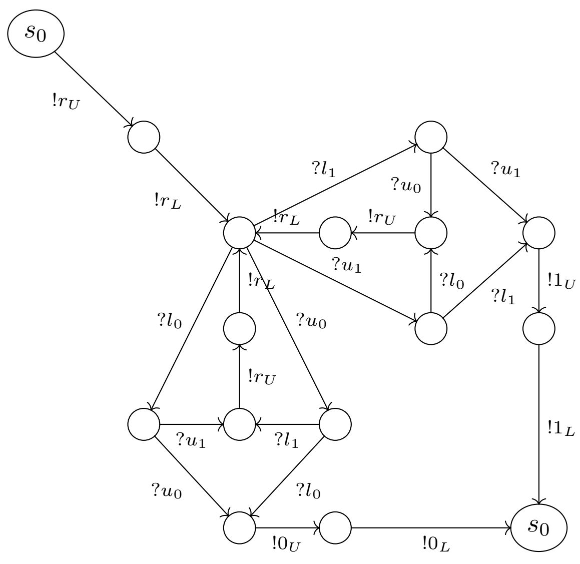

FIGURE 1. IOTS M_0

case where it is partial. We will therefore use the term IOTS to denote an input-enabled IOTS; where an IOTS might not be input-enabled we use the term *partial IOTS*. A *path* is a sequence $(q_1, a_1, q_2)(q_2, a_2, q_3)\dots$ of consecutive transitions, whose label is the trace produce by removing all instances of τ from $a_1a_2\dots$. For example, M_0 has a path with label $!r_U!r_L?i_1$. An IOTS is *divergent* if it has a state from which there is an infinite path whose transitions are all internal transitions (have label τ); as usual we assume that any IOTS considered is not divergent. An IOTS is *output-divergent* if it can reach a state from which there is an infinite path that contains outputs and internal actions only. An IOTS s is said to be *finitely-branching* if for every state q of s there are only finitely many transitions with starting state q.

Given IOTS $s = (Q, I, O, T, q_0)$ we use the following notation.

Definition 2.2. *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 $q \overset{a}{\Longrightarrow} q'$, for $a \in \mathcal{A}ct$, if there exist $q_1, \dots, q_{k+1}$ and $i \geq 0$ such that $q = q_1$, $q' = q_{k+1}$, $q_1 \xrightarrow{\tau} q_2, \dots, q_{i-1} \xrightarrow{\tau} q_i$, $q_i \xrightarrow{a} q_{i+1}$, $q_{i+1} \xrightarrow{\tau} q_{i+2}, \dots, q_k \xrightarrow{\tau} q_{k+1}$.
3. We write $q \overset{\epsilon}{\Longrightarrow} q'$ if there exist $q_1, \dots, q_k$, for $k \geq 1$, such that $q = q_1$, $q' = q_k$, $q_1 \xrightarrow{\tau} q_2, \dots, q_{k-1} \xrightarrow{\tau} q_k$.
4. We write $q \overset{\sigma}{\Longrightarrow} q'$ for $\sigma = a_1 \dots a_k \in \mathcal{A}ct^*$ if there exist $q_1, \dots, q_{k+1}$, $q = q_1$, $q' = q_{k+1}$ such that for all $1 \leq i \leq k$ we have that $q_i \overset{a_i}{\Longrightarrow} q_{i+1}$.
5. If $q_0 \overset{\sigma}{\Longrightarrow} q$ for some $\sigma \in \mathcal{A}ct^*$ then q is said to be reachable.
6. Given $\sigma \in \mathcal{A}ct^*$, we write $q \overset{\sigma}{\Longrightarrow}$ if there exists q' such that $q \overset{\sigma}{\Longrightarrow} q'$ and we say that σ is a trace *of s if $q_0 \overset{\sigma}{\Longrightarrow}$. We let $\mathcal{T}r^*(s)$ denote the set of (finite) traces of s.*
7. Given $\sigma = a_1a_2\dots \in \mathcal{A}ct^\omega$, we write $q \overset{\sigma}{\Longrightarrow}$ if there exist $q_1, q_2, \dots$ with $q_1 = q$ such that for all $1 \leq i$ we have that $q_i \overset{a_i}{\Longrightarrow} q_{i+1}$. We then say that σ is an infinite trace *of s if $q_0 \overset{\sigma}{\Longrightarrow}$. We let $\mathcal{T}r^\omega(s)$ denote the set of infinite traces of s.*
8. We say that trace σ of s is a quiescent trace *if there is a quiescent state q of s such that $q_0 \overset{\sigma}{\Longrightarrow} q$.*

We can define the projection function π_p, such that $\pi_p(\sigma)$ is the sequence of observations made at port p if the SUT produces $\sigma \in \mathcal{A}ct^* \cup \mathcal{A}ct^\omega$. For example, $!r_U!r_L?l_0$ is a trace of M_0 and this has projections $\pi_L(!r_U!r_L?l_0) = !r_L?l_0$ and $\pi_U(!r_U!r_L?l_0) = !r_U$. The projection function is defined as follows.

1. $\pi_p(\epsilon) = \epsilon$
2. If $a \in \mathcal{A}ct_p$ then $\pi_p(a\sigma) = a\pi_p(\sigma)$
3. If $a \in \mathcal{A}ct \setminus \mathcal{A}ct_p$ then $\pi_p(a\sigma) = \pi_p(\sigma)$

In distributed testing, if the SUT produces trace σ then the tester at $p \in \mathcal{P}$ observes $\pi_p(\sigma)$. Thus, two global traces σ and σ' are *observationally equivalent* if and only if they have the same set of projections and this is denoted $\sigma \sim \sigma'$. Thus, $\sigma \sim \sigma'$ if $\pi_p(\sigma) = \pi_p(\sigma')$ for all $p \in \mathcal{P}$. For example, the traces $!r_U!r_L?l_0?u_0$ and $!r_U!r_L?u_0?l_0$ of M_0 are observationally equivalent since they both have projection $!r_U?u_0$ at U and projection $!!r_L?l_0$ at L.

3. IMPLEMENTATION RELATIONS AND VERDICTS

In distributed testing, the local tester at port p only observes events at p. If the testers can ensure that they stop making observations at the same time then between them they have observed the set of local projections of a global trace of the SUT. Unfortunately, often it is not possible to ensure that the testers have all stopped making observations at the same time since this requires physically distributed entities to synchronise: some degree of synchronisation may be possible through the testers exchanging messages but message latency introduces imprecision into this. However, the testers can certainly know that they have all observed *prefixes* of projections of a common global trace of the SUT. This leads to our notion of an observation regarding the SUT.

Definition 3.1. *Given SUT r, $(\sigma_1, \dots, \sigma_m)$ is an* observation of r *if there exists a global trace $\sigma \in \mathcal{T}r^*(r)$ such that for all $p \in \mathcal{P}$, $\sigma_p \in pref(\pi_p(\sigma))$. We let $Obs(r)$ denote the set of possible observations of r.*

Recent work has independently considered a similar notion of an observation, with such an observation being called a multi-trace [21]. However, multi-traces were defined for the situation in which there are multiple components and the messages between the components are observed.

Given an observation $obs \in Obs(r)$ made in distributed testing, a specification s and implementation relation imp, we want to know whether obs is allowed by s under imp. If we have a decision procedure for determining whether obs is allowed by s under imp then this defines a Test Oracle and, equivalently, a *verdict function* [40]. We therefore say that a verdict function is a mapping from $Obs(r)$ to $\{pass, fail\}$. Given specification s and observation obs, ideally we want a verdict function to map obs to $pass$ if and only if there is some IOTS r' such that obs is an observation of r' and r' conforms to s under the implementation relation used.

As discussed in the literature on formal testing, ideally we want testing to have the following properties: it can never declare a correct implementation to be faulty (testing is sound); and if the SUT is faulty then testing can demonstrate this since the SUT will fail some test case (testing is exhaustive/complete). We can define similar properties for verdict functions.

Definition 3.2. *Let us suppose that we have verdict function v for specification s and implementation relation imp. We say that v is* sound *if for all r such that r imp s and observation $obs \in Obs(r)$ we have that $v(obs) = pass$. Further, v is* exhaustive *if for all r such that r does not conform to s under imp, there exists an observation $obs \in Obs(r)$ for which we have that $v(obs) = fail$.*

It is crucial that verdict functions are sound: otherwise we might declare a correct SUT to be faulty. However, we would also like them to be exhaustive since otherwise it might not be possible for testing to show that a particular faulty SUT is non-conforming.

We now give implementation relations that have been defined for distributed testing [16].

Definition 3.3. *Let r, s be IOTSs with the same input and output alphabets. We write r $\mathbf{dioco}_\delta$ s if for every trace $\sigma \in \mathcal{T}r^*(r)$ such that $\sigma\delta \in \mathcal{T}r^*(r)$ we have that there exists a trace $\sigma' \in \mathcal{T}r^*(s)$ such that $\sigma' \sim \sigma$.*

The definition of $\mathbf{dioco}_\delta$ refers to quiescent traces of the SUT (traces that end in quiescence) since we can know that the projections observed are all projections of the same global trace of the SUT and so we can compare the set of projections observed with global traces of the specification. If we consider M_0 then the observation $!r_U!r_L$ takes the model to a quiescent state. If in testing the testers at U an L observe $!r_U$ and $!r_L$ respectively and this is followed by quiescence then the testers can conclude that the SUT produced a global trace that has projection $!r_U$ at U and projection $!r_L$ at L. In contrast, if the local testers are allowed to stop recording observations before quiescence occurs then the local traces observed need not be projections of a single global trace. For example, if the local tester at U stops testing before $!r_U$ is observed and the local tester at L observes $!r_L$ then we have two projections of different global traces.

Note that quiescent traces of the SUT and specification are compared under $\sim$ and thus we require that if the SUT produced quiescent trace σ then the specification has a quiescent trace with the same projections. We might, instead, have considered the implementation relation **p-dioco** under which it is sufficient for the projection of σ at p to be identical to the projection at p of some quiescent trace of the specification [16] (for all p). However, **p-dioco** is strictly weaker than **dioco** since, for example, under **p-dioco** the SUT would be allowed to produce the quiescent trace $!o_1!o_2'$ even if the specification does not have global traces equivalent to this under $\sim$ but instead has quiescent traces $!o_1!o_2$ and $!o_1'!o_2'$. If observations made at different ports might later be brought together then one should use **dioco** rather than **p-dioco** even though the use of **dioco** requires the local testers to log their observations and for these logs to be brought together after testing finishes.

It has been observed that sometimes the SUT can follow a path where it never reaches a quiescent state, or after some quiescent state it does not reach another quiescent state. This is the case if the SUT has an infinite path from some state where all of the transitions on this path are labelled with outputs or τ (the SUT is output-divergent). The implementation relation $\mathbf{dioco}_\delta$ is not suitable in such situations since some behaviours of the SUT are not considered (the traces that cannot later be followed by quiescence). As a result of this, $\mathbf{dioco}_\delta$ has been generalised to processes that might be output-divergent and this was achieved by defining an implementation relation in terms of (some of) the infinite traces of the SUT [16]. In the following, a *run* of a process is an infinite trace that includes only finitely many inputs and for process s we have that $\mathcal{R}(s)$ is the set of runs of s.

Definition 3.4. *Let r, s be IOTSs. We write r **dioco** s if and only if for all $\sigma \in \mathcal{R}(r)$ there exists some $\sigma' \in \mathcal{R}(s)$ such that $\sigma \sim \sigma'$.*

Importantly, every finite trace of a process r is a prefix of a run of r since in any state where output is not enabled we can observe δ arbitrarily many times. Thus, in contrast to $\mathbf{dioco}_\delta$, every finite trace of the SUT is considered under **dioco**.

The restriction to infinite traces of r that contain only finitely many inputs is important [16]. To see why, consider an SUT that can initially loop with input $?i_1$ and can also produce output $!o_1$ from the initial state. Then one of its infinite traces is the sequence that contains only (infinitely many) $?i_1$. However, this trace corresponds to the repeated input of $?i_1$ blocking the output of $!o_1$ and so is not consistent with the standard assumption, for testing from IOTSs, that the environment/tester cannot block output. Naturally this restriction, that we only consider traces that contain finitely many inputs, is consistent with testing.

4. DEFINING VERDICT FUNCTIONS

The definitions of $\mathbf{dioco}_\delta$ and **dioco** are relatively simple and have the nice property that they correspond to relationships between languages. This has the benefit that we can use theory, results, and algorithms developed for formal languages. However, in distributed testing each tester observes a local trace and there is the need to decide upon a verdict. For $\mathbf{dioco}_\delta$, there is the potential to use quiescence to know that the local testers stopped testing at the same point. We can then compare the set of projections of a quiescent trace of the SUT with the projections of the quiescent traces of the specification. Thus, there is a well defined procedure for deciding whether a particular type of observation made in testing is allowed under $\mathbf{dioco}_\delta$, although deciding this is an NP-complete problem [35]. However, the testers might not stop making observations in a common quiescent state, and verdict functions have not been defined for this situation. In particular, if the SUT is output-divergent then it has states from which it is possible to take an infinite sequence of transitions such that the label of a transition from this sequence is either an output or τ. For such systems we may have that some traces of the SUT are not prefixes of quiescent traces and so cannot be assessed in the manner described above[4]. We also require a method for assigning verdicts for **dioco**; this certainly cannot involve comparing infinite traces of the SUT and specification since we do not observe infinite traces in testing. We therefore propose the following, alternative, approach to defining a verdict function.

Definition 4.1. *Observation $obs = (\sigma_1, \ldots, \sigma_m)$ is* allowed by *s if and only if there is a trace $\sigma \in \mathcal{T}r^*(s)$ such that $\sigma_p \in pref(\pi_p(\sigma))$ for all $p \in \mathcal{P}$. Given specification s, we call the corresponding verdict function v_s and so if $obs = (\sigma_1, \ldots, \sigma_m)$ then $v_s(obs) = pass$ if and only if there is a trace $\sigma \in \mathcal{T}r^*(s)$ such that $\sigma_p \in pref(\pi_p(\sigma))$ for all $p \in \mathcal{P}$.*

The idea simply is that although the testers do not know that the local traces they have observed are all projections of the same global trace of the SUT, they do know that they are all prefixes of projections of some global trace of the SUT. A verdict function v_s is a function from the set of observations to the set of verdicts and the above requires that all observations made regarding the SUT are also observations that could be made when interacting with the specification. We can now define an alternative implementation relation on the basis of the above: it essentially says that an SUT conforms to a specification if and only if all observations regarding the SUT are also observations regarding the specification.

Definition 4.2. *Given IOTSs r and s with the same input and output alphabets, we write r $\mathbf{dioco}_o$ s if and only if $Obs(r) \subseteq Obs(s)$.*

We will now prove that v_s defines a suitable verdict function for both $\mathbf{dioco}_\delta$ and **dioco**. We first consider specifications that are not output-divergent and the corresponding implementation relation $\mathbf{dioco}_\delta$, showing that in this case the verdict function is both sound and exhaustive. For the verdict function to be *sound* we require that all observations of correct SUTs are mapped to *pass*; for it to be *exhaustive* we require that if the SUT r does not conform to the specification s then some possible observation of r is mapped to $fail$. However, first we prove the following Lemma.

Lemma 4.1. *Given global trace $\sigma\delta$ and IOTS r, if r allows $(\pi_1(\sigma\delta), \ldots, \pi_m(\sigma\delta))$ then r has a global trace that is equivalent to $\sigma\delta$ under $\sim$.*

Proof. Since r allows $(\pi_1(\sigma\delta), \ldots, \pi_m(\sigma\delta))$ we have that r has some global trace σ' such that for all $p \in \mathcal{P}$, $\pi_p(\sigma\delta)$ is a prefix of $\pi_p(\sigma')$. Let σ' be some minimal such trace. Since each $\pi_p(\sigma\delta)$ ends in δ, by the minimality of σ' we know that σ' ends in δ. Further, from the definition of the projection function π_p we must have that σ and σ' have the same number of instances of δ. The result therefore follows. □

We now prove that our verdict function is sound and exhaustive for $\mathbf{dioco}_\delta$.

Proposition 4.1. *Let r, s be IOTSs that are not output-divergent. We have r $\mathbf{dioco}_\delta$ s if and only if every observation of r is allowed by s.*

Proof. We will start by proving left-to-right and so assume that r $\mathbf{dioco}_\delta$ s. Let us suppose that $(\sigma_1, \ldots, \sigma_m)$ is an observation of r. Since r is not output-divergent, there exists a quiescent trace $\sigma \in \mathcal{T}r^*(r)$ such that for all $p \in \mathcal{P}$ we have that $\sigma_p \in pref(\pi_p(\sigma))$. Further, since r $\mathbf{dioco}_\delta$ s, there exists some quiescent trace $\sigma' \in \mathcal{T}r^*(s)$ such that $\sigma' \sim \sigma$. Thus, for all $p \in \mathcal{P}$ we have that $\pi_p(\sigma') = \pi_p(\sigma)$ and so $(\sigma_1, \ldots, \sigma_m)$ is allowed by s as required.

We now prove the right-to-left result and assume that all observations of r are allowed by s. Let σ be a quiescent trace of r: it is sufficient to prove that s has a quiescent trace that is equivalent to σ under $\sim$. Since σ is a quiescent trace of r, we have that $(\pi_1(\sigma\delta), \ldots, \pi_m(\sigma\delta))$ is an observation of r and this must be allowed by s. Thus, by Lemma 4.1, s has a global trace that is equivalent to σ under $\sim$. The result therefore follows. □

The following is an immediate consequence of the above result.

Corollary 4.1. *Given IOTSs r and s that are not output-divergent, r **dioco** s if and only if r $\mathbf{dioco}_o$ s.*

We now consider the implementation relation **dioco**, which is defined for cases where IOTSs might be output-

[4]This possibility motivated the use of infinite traces in the definition of **dioco**.

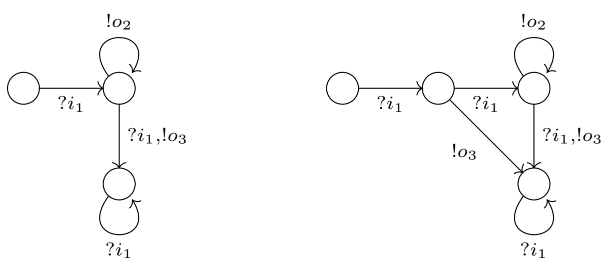

FIGURE 2. IOTSs s and r

divergent. First, we prove that our verdict function is sound for this.

Proposition 4.2. *Given IOTSs r, s, if r* **dioco** *s then every observation of r is allowed by s.*

Proof. Let us suppose that $(\sigma_1, \ldots, \sigma_m)$ is an observation of r. By definition, there exists a finite trace $\sigma \in \mathcal{T}r^*(r)$ such that for all $p \in \mathcal{P}$ we have that $\sigma_p \in pref(\pi_p(\sigma))$. Further, we have that σ is a prefix of a run σ_1 of r. Since r **dioco** s there exists some run σ_1' of s such that $\sigma_1' \sim \sigma_1$. Since σ is a prefix of σ_1 we have that $\sigma_p \in pref(\pi_p(\sigma_1))$ for all $p \in \mathcal{P}$. Thus, since $\sigma_1' \sim \sigma_1$ we have that $\pi_p(\sigma_1') = \pi_p(\sigma_1)$ for all $p \in \mathcal{P}$ and so $\sigma_p \in pref(\pi_p(\sigma_1'))$ for all $p \in \mathcal{P}$. As a result, there exists a finite prefix σ' of σ_1' such that $\sigma_p \in pref(\pi_p(\sigma'))$ for all $p \in \mathcal{P}$. Thus, since σ' is a trace of s we have that $(\sigma_1, \ldots, \sigma_m)$ is allowed by s as required. □

The question now is whether our verdict is exhaustive for **dioco**.

Example 1. Consider the IOTSs shown in Figure 2 that have three ports and only one input $?i_1$. It is straightforward to see that they are input-enabled.

Consider the finite traces of r that are *not* traces of s. These are of the following forms:

- Traces of the form $?i_1\sigma_1\sigma_2$ for σ_1 in the regular language defined by $?i_1!o_2^*$ and σ_2 in the language defined by $?i_1^*$. Such a trace is equivalent under $\sim$ to the trace $\sigma_1\sigma_2?i$ of s.
- Traces of the form $?i_1\sigma_1\sigma_2$ for σ_1 in the language defined by $?i_1!o_2^*$ and σ_2 in the language defined by $!o_3?i_1^*$. Such a trace is equivalent under $\sim$ to the trace $\sigma_1\sigma_2?i$ of s.

As a result, all finite traces of r are equivalent to finite traces of s under $\sim$. However, the infinite trace $?i_1?i_1!o_2^\omega$ of r is not equivalent to any trace of s under $\sim$. Thus, we have that r does not conform to s under **dioco** but in distributed testing we cannot demonstrate this using finite observations.

As a result, we know that no verdict function that only uses finite observations can be exhaustive for **dioco**. We therefore obtain the following result.

Theorem 4.1. *Implementation relation* **dioco** *is strictly stronger than* **dioco**$_o$*.*

Since **dioco**$_o$ is defined exactly in terms of the observations made in testing, this result suggests that **dioco** is too strong for processes that are output-divergent. Thus, **dioco**$_o$ is more suitable that **dioco** for testing, though it may well be that **dioco** is useful for other activities such as refinement.

If we consider the above example we can see that the run $?i_1!o_2^\omega$ that distinguishes the processes has a port p at which there are only finitely many events.

Definition 4.3. *An IOTS s* allows starvation *if it has some run σ and port p such that $\pi_p(\sigma)$ is finite.*

Here it is important that we consider runs of s and not all infinite traces of s since every input-enabled IOTS that has one or more inputs contains an infinite trace whose projection is empty at some port: to construct such an infinite trace it is sufficient to choose an input $?i_p$ at some port p and use $?i_p^\omega$.

Since a run contains only finitely many inputs we have the following property.

Proposition 4.3. *IOTS s allows starvation if and only if it has a reachable state q from which there is an infinite path whose label σ contains no inputs and such that there is a port p such that $\pi_p(\sigma) = \epsilon$.*

Proof. First assume that s allows starvation and so it has an infinite path ρ with label σ that contains only finitely many inputs such that there is a port p where $\pi_p(\sigma)$ is finite. Then $\rho = \rho_1\rho_2$ for some minimal path ρ_1 that contains every element of $\pi_p(\sigma)$ and all of the inputs from σ. Clearly, ρ_2 starts at a reachable state and has a label such that $\pi_p(\sigma) = \epsilon$ as required.

Now let us suppose that s has a reachable state q from which there is an infinite path ρ whose label σ contains only outputs and such that $\pi_p(\sigma) = \epsilon$. Since q is reachable there is some finite path ρ_1 that reaches q. As a result, $\rho_1\rho$ is an infinite path of s such that the label $\sigma_1\sigma$ has finite projection at p. Further, $\sigma_1\sigma$ has only finitely many inputs and so is a run. Thus, s allows starvation as required. □

Thus, processes that allow starvation have a port p and state q from which they can continue to produce output indefinitely without p making any observations. Below we prove that if the specification does not allow starvation then the verdict function for **dioco** is exhaustive and so **dioco** and **dioco**$_o$ coincide. The proof will use König's Lemma which is the following (as quoted in [41]).

Lemma 4.2. *An infinite tree in which every node has finite arity contains an infinite path.*

We now prove that v_s is exhaustive for **dioco** if the SUT and the specification s are finitely-branching and do not allow starvation.

Proposition 4.4. *Let r, s be* IOTSs *that do not allow starvation and are finitely-branching. If every observation of r is allowed by s then r* **dioco** *s.*

Proof. We assume that every observation of r is allowed by s and it is sufficient to prove that given run σ of r, σ is equivalent under $\sim$ to a run of s. We will use proof by contradiction and suppose that no run of s is equivalent to σ under $\sim$. For $i \geq 1$ let σ_i denote the prefix of σ with length i. Since every observation of r is allowed by s, for every σ_i we have that $(\pi_1(\sigma_i), \ldots, \pi_m(\sigma_i))$ is allowed by s. Further, for all p we know that $\pi_p(\sigma_i)$ is a prefix of $\pi_p(\sigma)$.

Let σ'_i denote a longest trace of s such that for all p we have that $\pi_p(\sigma'_i)$ is a prefix of $\pi_p(\sigma_i)$. Since $(\pi_1(\sigma_i), \ldots, \pi_m(\sigma_i))$ is allowed by s, the length of σ'_i is at least that of the shortest $\pi_p(\sigma_i)$. Let s_σ denote the acyclic graph produced when we unfold the IOTS formed by restricting s as follows: identify the set of paths of s that start at the unique initial state of s and have label σ'_i for some i and retain only transitions that occur in these paths. Since r does not allow starvation, and the length of σ'_i is at least the minimum of the lengths of the $\pi_p(\sigma_i)$, there is no upper bound on the lengths of the σ'_i. Thus, s_σ is a tree with finite arity (since s is finitely-branching) and so by König's Lemma (Lemma 4.2) we know that s_σ contains an infinite path ρ. Let σ' denote the label of ρ. Now consider a port $p \in \mathcal{P}$ and the projection $\pi_p(\sigma')$. If $\pi_p(\sigma') \neq \pi_p(\sigma)$ then there is some finite prefix σ'' of σ' such that $\pi_p(\sigma'')$ is not a prefix of $\pi_p(\sigma)$. But, by definition, σ'' must be a prefix of some σ'_i and this contradicts the definition of the σ'_i. This provides a contradiction as required and so the result follows. □

We therefore obtain the following.

Theorem 4.2. *Given* IOTSs *r, s that do not allow starvation and are finitely-branching, r* **dioco** *s if and only if r* **dioco***$_o$ s.*

To summarise, for **dioco** the verdict function is sound but need not be exhaustive. However, if processes are finitely-branching and do not allow starvation then the verdict function is both sound and exhaustive for **dioco**. It is natural to ask whether the verdict function is as close to being exhaustive as it could be for specifications that allow starvation. This requires that not only can a non-conforming SUT fail a test but also that it fails any tests where the observation made is not allowed by any conforming `IOTS`. The following shows that it is since if there is a finite trace of the SUT that cannot be extended to form an infinite trace that is equivalent to one in $\mathcal{T}r^\omega(s)$ then there are observations of r that lead to verdict *fail*.

Proposition 4.5. *Let r, s be* IOTSs *and let us suppose that r has a finite trace σ such that for every run $\sigma' \in \mathcal{A}ct^\omega$ we have that $\sigma\sigma'$ is not equivalent to any infinite trace of s. Then observation $(\pi_1(\sigma), \ldots, \pi_m(\sigma))$ of r is given verdict fail.*

Proof. Proof by contradiction: assume that σ is a finite trace of r, observation $(\pi_1(\sigma), \ldots, \pi_m(\sigma))$ is given verdict *pass*, and for every run $\sigma' \in \mathcal{A}ct^\omega$ we have that $\sigma\sigma'$ is not equivalent to any infinite trace of s. Since $(\pi_1(\sigma), \ldots, \pi_m(\sigma))$ is given verdict *pass* we have that $(\pi_1(\sigma), \ldots, \pi_m(\sigma))$ is an observation allowed by the specification. But this means that the specification has a finite trace σ_s such that $\pi_p(\sigma) \in \mathit{pref}(\pi_p(\sigma_s))$ for all p. Further, σ_s can be extended to a run σ'_s of s. By the definition of σ'_s we have that $\pi_p(\sigma) \in \mathit{pref}(\pi_p(\sigma'_s))$ for all p. Thus, there exists some σ' such that $\sigma\sigma' \sim \sigma'_s$. Finally, since σ'_s is a run we also have that σ' is a run. This provides a contradiction as required and so the result follows. □

The implementation relation **dioco**$_o$ is defined for IOTSs that are input-enabled. It is natural to ask how we can relax this for the case where an IOTS need not be input-enabled. In order to do so for an IOTS that represents the SUT we need to give a semantics for an IOTS being partial and typically this involves concepts such as *refusals* (the observation of the SUT refusing an input). In this paper we do not include refusals as observations and we follow the **ioco** approach of assuming that the SUT is input-enabled; the extension to partial SUTs is a topic for future work. We now consider the case where the specification might be partial but the SUT is input-enabled.

One approach to providing a semantics for partial specifications is to say that if an input $?i$ is received when the specification is in a state q where there is no transition with label $?i$ then all behaviours are allowed after $?i$. We take this approach here and define a corresponding closure operation on partial IOTSs as follows.

Definition 4.4. *If $s = (Q, I, O, T, q_0)$ is a partial IOTS then the closure of s, denoted $\mathcal{C}(s)$, is the IOTS $(Q \cup \{q_c, q'_c\}, I, O, T', q_0)$ in which T' is T with the following transitions added:*

- *Given state q and $?i \in I$, if there is no transition from q with label $?i$ then we add the transition $(q, ?i, q_c)$.*
- *For all $a \in I \cup O$, we add the transition (q_c, a, q_c).*
- *We add the transitions (q_c, τ, q'_c) and (q'_c, δ, q'_c).*
- *For all $?i \in I$, we add the transition $(q'_c, ?i, q_c)$.*

The basic idea is that if $?i$ is received in a state q for which there is no corresponding transition then $?i$ takes $\mathcal{C}(s)$ to state q_c and from this state there are self-loops for all inputs and outputs. From state q_c it is possible to take a τ transition to a state q'_c where it is possible to observe δ but from which we cannot observe an output without first receiving an input. The use of q'_c as well as q_c is to ensure that we do not include traces that follow δ with an output; such traces are not valid observations.

We then obtain the implementation relation **dioco**^p_o.

DEFINITION 4.5. *Given partial IOTS s and IOTS r, r $\mathbf{dioco}_o^p$ s if and only if $Obs(r) \subseteq Obs(\mathcal{C}(s))$.*

The following is clear from the definition.

PROPOSITION 4.6. *Given partial IOTS s and IOTS r, r $\mathbf{dioco}_o^p$ s if and only if r $\mathbf{dioco}_o$ $\mathcal{C}(s)$.*

Thus, if we can solve the Oracle problem for $\mathbf{dioco}_o$ then we can also solve the Oracle problem for $\mathbf{dioco}_o^p$; we simply generate $\mathcal{C}(s)$ and then apply the verdict function for $\mathbf{dioco}_o$. In addition, the closure does not significantly increase the size of the specification. As a result, we will focus on the case where the specification is input-enabled.

König's Lemma (Lemma 4.2) was used to prove the following well-known result [41] (Lemma 2.1).

PROPOSITION 4.7. *Given finitely-branching IOTS s and infinite sequence σ, if all proper prefixes of σ label paths of s then σ labels a path of s.*

Proposition 4.4 suggests that we might be able to generalise Proposition 4.7 to the case where we are interested in the set of runs that are equivalent to runs of s under $\sim$ and we now show how this can be done. First we will define the languages produced by applying a commutation operator to $\mathcal{T}r^*(s)$ and $\mathcal{R}(s)$.

DEFINITION 4.6. *Given IOTS s, languages $\mathcal{L}(s)$ and $\mathcal{L}^\omega(s)$ are defined as follows.*

$$\mathcal{L}(s) = \{\sigma \in \mathcal{A}ct^* | \exists \sigma' \in \mathcal{T}r^*(s). \sigma \sim \sigma'\}$$

$$\mathcal{L}^\omega(s) = \{\sigma \in \mathcal{A}ct^\omega | \exists \sigma' \in \mathcal{R}(s). \sigma \sim \sigma'\}$$

We now show that a version of Lemma 4.2 holds for $\mathcal{L}(s)$ and $\mathcal{L}^\omega(s)$.

PROPOSITION 4.8. *Given `IOTS` s that is finitely-branching and does not allow starvation and run $\sigma \in \mathcal{A}ct^\omega$, if every finite prefix of σ is in $\mathcal{L}(s)$ then σ is in $\mathcal{L}^\omega(s)$.*

Proof. For $i \geq 1$ let σ_i denote the prefix of σ with length i. Since $\sigma_i \in \mathcal{L}(s)$ there exists some $\sigma'_i \in \mathcal{T}r^*(s)$ such that $\sigma'_i \sim \sigma_i$. Let s_σ denote the acyclic graph produced when we unfold the IOTS formed by restricting s as follows: identify the set of paths of s that start at the unique initial state of s and have label σ'_i for some i and retain only transitions that occur in these paths. There is no upper bound on the lengths of the σ'_i. Thus, s_σ has an infinite number of states and is finitely-branching and so by König's Lemma (Lemma 4.2) we know that s_σ contains an infinite path ρ. Let σ' denote the label of ρ. Now consider a port $p \in \mathcal{P}$ and the projections $\pi_p(\sigma')$ and $\pi_p(\sigma)$. Clearly σ' is an infinite trace of s and has no more inputs that σ and so σ' is a run of s. Since no infinite trace being considered contains only finitely many events at a port, we have that $\pi_p(\sigma')$ is infinite and so $\pi_p(\sigma')$ is not a proper prefix of $\pi_p(\sigma)$. Thus, if $\pi_p(\sigma') \neq \pi_p(\sigma)$ then there is some finite prefix σ'' of σ' such that $\pi_p(\sigma'')$ is not a prefix of $\pi_p(\sigma)$. But, by definition, σ' must be a prefix of some σ'_i and this contradicts the definition of the σ'_i. This provides a contradiction as required and so the result follows. □

5. THE COMPLEXITY OF THE ORACLE PROBLEM

We have defined a verdict function for distributed testing and proved that it has the expected properties. This section explores the complexity of the associated Oracle problem, of deciding whether an observation is allowed. We prove that this problem is NP-complete but can be solved in polynomial time if we have an upper bound on the number of ports. Previous work has shown that the Oracle problem for finite state machines is NP-hard [35] but this is different from the problem considered here since finite state machines are quiescent after each input/output pair and so we essentially use $\mathbf{dioco}_\delta$.

We will relate the Oracle problem to the one-in-three SAT problem.

DEFINITION 5.1. *Given boolean variables $z_1, \ldots, z_r$ let $C_1, \ldots, C_k$ denote sets of three literals, where each literal is either a variable z_i or its negation. The one-in-three SAT problem is: does there exist an assignment to the boolean variables such that each C_i contains exactly one true literal.*

The one-in-three SAT problem is NP-complete [42].

The proof that the Oracle problem is NP-hard will take an instance of the one-in-three SAT problem and construct an IOTS s and observation $obs = (\sigma_1, \ldots, \sigma_m)$ such that obs is allowed under $\mathbf{dioco}_o$ if and only if the instance of the one-in-three SAT problem has a solution. Let us suppose that the instance of the one-in-three SAT problem has boolean variables $z_1, \ldots, z_r$ and clauses $C_1, \ldots, C_k$. The IOTS s will have initial state q_{in} and for each variable z_x it will have a corresponding input $?i_x$ at port x. Further, for each cause C_i, s will have a unique output $!o_{k+i}$ at port $k+i$. As a result, the inputs and outputs are at different ports. The 'core' of the IOTS has, for each input $?i_x$, two cycles in state q_{in}: one cycle (ρ_x^T) has input $?i_x$ followed by output $!o_{k+i}$ for all i such that C_i has literal z_x; the other cycle (ρ_x^F) has input $?i_x$ followed by output $!o_{k+i}$ for all i such that C_i has literal $\neg z_x$. Before explaining the construction further, we give a simple example in which we have four boolean variables z_1, z_2, z_3, z_4 and clauses $C_1 = z_1 \wedge z_3 \wedge \neg z_4$, $C_2 = z_2 \wedge \neg z_3 \wedge \neg z_4$, and $C_3 = z_1 \wedge z_2 \wedge z_4$. The construction explained above leads to the partial IOTS outlined in Figure 3 in which the central state is the initial state.

Now let us suppose that the input of $?i_1?i_2?i_3?i_4$ in this examples leads to path $\rho_1^T \rho_2^T \rho_3^F \rho_4^T$. Then the resultant quiescent trace is $?i_1!o_5!o_7?i_2!o_6!o_7?i_3!o_6?i_4!o_7$ and so to the observation $(?i_1, ?i_2, ?i_3, ?i_4, !o_5, !o_6 !o_6, !o_7 !o_7 !o_7)$. From this we can see that if z_1, z_2, z_4 take on the value of true and z_3

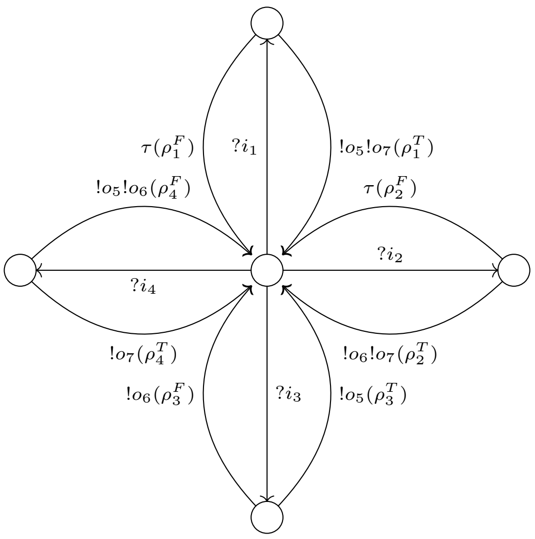

FIGURE 3. Finite State Machine M_0

takes on the value of false then: C_1 contains one true literal (since the observation has one instance of $!o_5$); C_2 contains two true literals (since the observation has two instances of $!o_6$); and C_3 contains three true literals (since the observation has three instances of $!o_7$). More generally, if the input of $?i_1?i_2?i_3?i_4$ leads to paths of the partial IOTS being followed then it *can* lead to an observation of the form $(?i_1, ?i_2, ?i_3, ?i_4, !o_5^{n_1}, !o_6^{n_2}, !o_7^{n_3})$ if and only if there is some assignment to the boolean variables $z_1, \ldots, z_4$ such that C_i contains n_i true literals $1 \leq i \leq 3$. More specifically, if the input of $?i_1?i_2?i_3?i_4$ leads to paths of the partial IOTS being followed then it *can* lead to an observation of the form $obs = (?i_1, ?i_2, ?i_3, ?i_4, !o_5, !o_6, !o_7)$ if and only if there is some assignment to the boolean variables $z_1, \ldots, z_4$ such that each C_i contains exactly one true literal (and so there is a solution to this instance of the one-in-three SAT problem). The proof essentially takes this partial IOTS and completes it in such a manner that this observation *obs* can only occur through a path in the partial IOTS being followed.

Lemma 5.1. *The problem of deciding whether an observation $obs = (\sigma_1, \ldots, \sigma_m)$ is allowed by an IOTS s that has finite sets of states, inputs and outputs is NP-hard.*

Proof. To show that the problem is NP-hard we will show that we can reduce an instance of the one-in-three SAT problem to this. We therefore suppose that we have variables $z_1, \ldots, z_r$ and clauses $C_1, \ldots, C_k$. We will define an IOTS s with $r+k$ ports, inputs $?i_1, \ldots, ?i_r$ at ports $1, \ldots, r$ and outputs $!o_{r+1}, \ldots, !o_{r+k}$ at ports $r+1, \ldots, r+k$.

From the initial state q_{in} of s, for each input $?i_x$ there are two cycles:

1. A cycle ρ_x^T that starts with $?i_x$ and, for all $1 \leq j \leq k$, has (one) output $!o_{r+j}$ at port $r+j$ if and only if C_j contains literal z_x and otherwise has no output at port $r+j$. For all $1 \leq p \leq r$ ρ_x^T has no output at port p.
2. A cycle ρ_x^F that starts with input $?i_x$, for all $1 \leq j \leq k$, has (one) output $!o_{r+j}$ at port $r+j$ if and only if C_j contains literal $\neg z_x$ and otherwise has no output at port $r+j$. For all $1 \leq p \leq r$ ρ_x^T has no output at port p.

We also add an 'error state' q_e and from every state except q_{in} all inputs take s to q_e. From q_e there are self-loop transitions labelled with inputs and also a self-loop transition labelled with 'error' output $!o_e$ at port 1 that is not produced by any other transition.

Now consider the observation $obs = (?i_1\delta, ?i_2\delta, \ldots, ?i_r\delta, !o_{r+1}\delta, \ldots, !o_{r+k}\delta)$. In this, each input is received once by s and each output of the form $!o_{r+l}$ is observed once. Since the observations all end in δ, the projections observed are all projections of the same global trace of s. Thus, *obs* is allowed by s if and only if s has a trace with these projections. Further, since $!o_e$ is not observed we must have that each input was received when s was in state q_{in}. Thus, since each input is received exactly once, *obs* is allowed by s if and only if it is the set of projections of the label of a path ρ of s that consists of loops from q_{in} and so $\rho \sim \rho_1^{B_1} \ldots \rho_r^{B_r}$ for some $B_1, \ldots, B_r \in \{true, false\}$. By the definitions of the $\rho_j^{B_j}$, this is the case if and only if the assignment $z_j = B_j$ for all $1 \leq j \leq r$ is a solution to the instance of the one-in-three SAT problem. Thus, this instance of the one-in-three SAT problem has a solution if and only if *obs* is allowed by s. The result thus follows from the one-in-three SAT problem being NP-hard and the fact that it is possible to construct s and *obs* in polynomial time. □

We now show that the problem is in NP by showing how a non-deterministic Turing machine can solve it in polynomial time.

Lemma 5.2. *Given IOTS s that has finite sets of states, inputs, and outputs, the problem of deciding whether an observation $obs = (\sigma_1, \ldots, \sigma_m)$ is allowed by an IOTS s is in NP.*

Proof. We will design a non-deterministic Turing Machine $\mathcal{T}$ that operates as follows. We will construct $\mathcal{T}$ so that it essentially guesses a path and checks *obs* against the label of this path. $\mathcal{T}$ stores the current state q of s and the part obs' of *obs* not yet processed in the path being guessed and so starts with these values being the initial state of s and *obs*. Thus, if the path current guessed has label σ and ending state q then $\mathcal{T}$ stores state q and the observation $obs' = (\sigma'_1, \ldots, \sigma'_m)$ such that (for port p): if $\pi_p(\sigma)$ is a prefix of σ_p then $\sigma_p = \pi_p(\sigma)\sigma'_p$ (σ'_p is the remaining local trace to be observed at port p); and otherwise we must have that all observations in σ_p have been made at p and so $\sigma'_p = \epsilon$. At each step $\mathcal{T}$ guesses a transition (q_1, a, q_2) of s where q_1 is the current state of s stored by $\mathcal{T}$. $\mathcal{T}$ then applies one of the following steps in which $a \in \mathcal{A}ct_p \cup \{\tau\}$ and $obs' = (\sigma'_1, \ldots, \sigma'_m)$ is the current observation stored (the projections of the label of the currently guessed path).

1. If $a = \tau$ then the values stored by $\mathcal{T}$ are updated as follows: q is assigned the value q_2 and obs' is not changed.
2. If $\sigma'_p = a\sigma''_p$ for some σ''_p then the values stored by $\mathcal{T}$ are updated as follows: q is assigned the value q_2 and obs' is assigned $(\sigma'_1, \ldots, \sigma'_{p-1}, \sigma''_p, \sigma'_{p+1}, \ldots, \sigma'_m)$. This case simply removes a from the sequence of observations still to be made at p and updates the current state. If $obs' = (\epsilon, \ldots, \epsilon)$ then $\mathcal{T}$ terminates with success and otherwise processing continues.
3. If $\sigma'_p = \epsilon$ then processing continues and the values stored by $\mathcal{T}$ are updated as follows: q is assigned the value q_2 and obs' remains unchanged. This is the case where the local trace σ_p in obs is already a prefix of the projection (on p) of the path being guessed.
4. If $\sigma'_p \neq \epsilon$ and σ'_p does not start with a then this execution of $\mathcal{T}$ terminates with failure. Here it has been determined that the path being guessed has a label whose projection at p does not start with σ_p.

We also have that $\mathcal{T}$ stores the set of states visited since obs' was last changed. If $\mathcal{T}$ visits a state q that has been met since obs' was last changed then it terminates with failure (this stops $\mathcal{T}$ from taking a cycle in which obs' does not change since such a cycle could be repeated, leading to non-termination).

We now show that $\mathcal{T}$ solves the Oracle problem. First, let us suppose that $obs = (\sigma_1, \ldots, \sigma_m)$ is an observation of s and so the verdict should be *pass*. There therefore exists a minimal trace σ of s such that $\sigma_p \in pref(\pi_p(\sigma))$ for all $1 \leq p \leq m$. Recall that a non-deterministic Turing Machine terminates with success if one of its possible executions ('guesses') terminates with success. Thus, $\mathcal{T}$ terminates with success since it can guess a minimal path of s that has trace σ. Now, consider the case where the verdict should be $fail$. Since s has finitely many states and $\mathcal{T}$ cannot visit a state of s more than once with the same obs', if obs is not an observation of s then $\mathcal{T}$ must terminate (it cannot follow an infinite sequence of steps) and terminates with failure. Thus, $\mathcal{T}$ solves the verdict problem.

Finally, consider the time complexity of $\mathcal{T}$. Since $\mathcal{T}$ cannot visit a state of s more than once with the same obs', the number of steps taken cannot exceed the number of states of s multiplied by the sum of the lengths of the local traces in obs. Thus, $\mathcal{T}$ terminates in polynomial time. We therefore have that the problem is in NP and so the result follows. □

Finally, we prove that the problem is NP-complete for `IOTSs` with finite sets of states and input/output alphabets.

Theorem 5.1. *Given IOTS s that has finite sets of states, inputs, and outputs, the problem of deciding whether an observation $obs = (\sigma_1, \ldots, \sigma_m)$ is allowed by s is NP-complete.*

Proof. This follows from Lemmas 5.1 and 5.2. □

We now consider the case where there is an upper bound on the number of ports. This is an interesting case since in practice there are normally only a few ports but the IOTS may have many states and the local traces observed may be relatively long.

Let us suppose that we have observation $obs = (\sigma_1, \ldots, \sigma_m)$ and for $p \in \mathcal{P}$ let $\sigma_p = a_1^p a_2^p \ldots a_{\ell_p}^p$. Then any trace that has a prefix whose projection at p is σ_p must be in the regular language $L_p(\sigma_p) = (\mathcal{A}ct \setminus \mathcal{A}ct_p)^* a_1^p (\mathcal{A}ct \setminus \mathcal{A}ct_p)^* \ldots (\mathcal{A}ct \setminus \mathcal{A}ct_p)^* a_{\ell_p}^p \mathcal{A}ct^*$. Thus, in order to determine whether obs is allowed by `IOTS` s it is sufficient to decide whether the language $\mathcal{T}r^*(s) \cap L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m)$ is empty. The following proves that this works.

Proposition 5.1. *Observation $obs = (\sigma_1, \ldots, \sigma_m)$ is allowed by `IOTS` s if and only if $\mathcal{T}r^*(s) \cap L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m) \neq \emptyset$.*

Proof. First let us suppose that $\sigma \in \mathcal{T}r^*(s) \cap L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m)$. Thus, σ is a trace of s. Further, for all $p \in \mathcal{P}$ we have that $\sigma \in L_p(\sigma_p)$ and so σ_p is a prefix of $\pi_p(\sigma)$. Thus, obs is allowed by s as required.

Now assume that obs is allowed by s and let σ be a finite trace of s such that for all $p \in \mathcal{P}$ we have that σ_p is a prefix of $\pi_p(\sigma)$. However, by the definition of $L_p(\sigma_p)$, this is the case if and only if $\sigma \in L_p(\sigma_p)$. Thus, $\sigma \in \mathcal{T}r^*(s)$ and for all $p \in \mathcal{P}$ we have that $\sigma \in L_p(\sigma)$. We therefore have that $\sigma \in \mathcal{T}r^*(s) \cap L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m)$ and so $\mathcal{T}r^*(s) \cap L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m) \neq \emptyset$ as required. □

In order to construct a finite automaton that accepts $\mathcal{T}r^*(s) \cap L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m)$ it is sufficient to take the product of the separate automata. The states of this product automaton are tuples of states of the individual finite automata and so the number of states of the product automaton can be the product of the numbers of states of the individual automata. Thus, the product automaton can take exponential space and the problem of deciding whether there is some common sequence in the languages of a set of finite automaton (the FA-Int problem) is PSPACE-complete [43]. However, we now show that if we have an upper bound on the number of ports m then we can produce a FA that represents the language $L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m)$ in polynomial time; a similar result has been shown for the corresponding problem when testing using **dioco** and IOTSs are not output-divergent [44].

In order to represent the language $L_1(\sigma_1) \cap \ldots \cap L_m(\sigma_m)$ we can construct an FA $\mathcal{M}(\sigma_1, \ldots, \sigma_m)$ as defined below. The essential idea is that a state of $\mathcal{M}(\sigma_1, \ldots, \sigma_m)$ is defined by a tuple $(x_1, \ldots, x_m)$ that represents the situation in which the first x_p elements of σ_p have been observed at port p and if $x_p = \ell_p$ then additional observations might have been made at p.

Definition 5.2. *Let us suppose that $\sigma_i = a_1^p \ldots a_{\ell_p}^p$, $1 \leq p \leq m$. Then $\mathcal{M}(\sigma_1, \ldots, \sigma_m)$ is the FA $(S, s_0, \mathcal{A}ct, h, F)$ in which:*

1. *S is the set of states defined by tuples of the form $(x_1, \ldots, x_m)$ in which for all $1 \leq p \leq m$ we have that $0 \leq x_p \leq \ell_p$.*
2. *s_0 is the initial state $(0, \ldots, 0)$.*
3. *$\mathcal{A}ct$ is the alphabet.*
4. *h is the transition relation defined by $((x_1, \ldots, x_m), a, (x'_1, \ldots, x'_m)) \in h$ for $a \in \mathcal{A}ct$ if and only if one of the following conditions hold:*
   - *(a) $a \in \mathcal{A}ct_p$, $x_p < \ell_p$, $a = a^p_{x_1+1}$, $x'_p = x_p + 1$ and for all $1 \leq q \leq m$ we have that if $p \neq q$ then $x'_q = x_q$.*
   - *(b) $a \in \mathcal{A}ct_p$, $x_p = \ell_p$, and for all $1 \leq q \leq m$ have that if $p \neq q$ then $x'_q = x_q$.*
5. *$F = \{(\ell_1, \ldots, \ell_m)\}$ is the set of final states.*

It is clear that the number of states of $\mathcal{M}(\sigma_1, \ldots, \sigma_m)$ is $(|\sigma_1| + 1)(|\sigma_2| + 1) \cdots (|\sigma_m| + 1)$ and so is polynomial in terms of the size of the problem if we have an upper bound on the number of ports. The problem then becomes one of deciding whether the intersection of the regular languages defined by s and $\mathcal{M}(\sigma_1, \ldots, \sigma_m)$ is non-empty, a problem that can be solved in time that is polynomial in the sizes of s and $\mathcal{M}(\sigma_1, \ldots, \sigma_m)$. We therefore obtain the following result.

Theorem 5.2. *If we have an upper bound on the number of ports then the problem of deciding whether an observation $obs = (\sigma_1, \ldots, \sigma_m)$ is allowed by an IOTS s can be solved in polynomial time.*

6. RELATED WORK

Distributed testing was initially explored in the context of testing from an FSM. This was in the area of protocol conformance testing in which there are two ports: the upper tester U interacts directly with the implementation of a layer of the protocol stack (by using its features) while the lower tester sits on another machine. The initial work identified the potential for distributed testing to introduce additional controllability problems [12, 4] and observability problems [5]. A controllability problem occurs when the tester at a port p cannot know when to supply an input since it did not observe events in the previous input/output pair. Let us suppose that the tester at port 1 should send input $?i_1$, this should lead to output $!o_1$ to port 1, and the tester at port 2 should then send $?i_2$. This scenario is shown in Figure 4 in which vertical lines represent processes, time progresses as we move down, and arcs represent messages. The tester at port 2 does not know when to send its input since it does not observe the previous input and output.

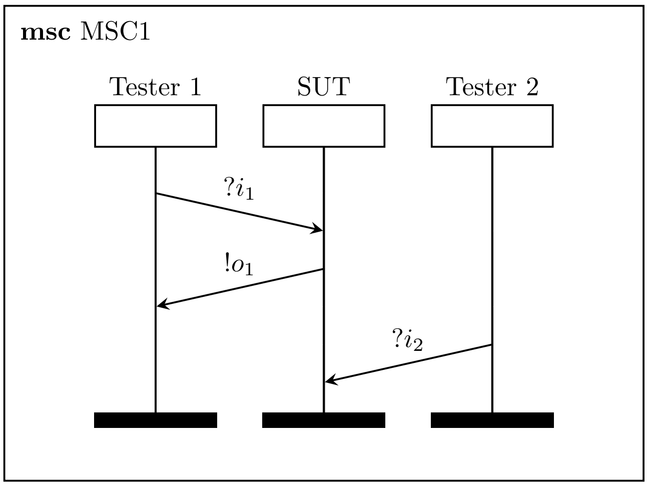

FIGURE 4. A controllability problem

An observability problem occurs if the global trace σ that occurred in testing is not a trace of the specification but is observationally equivalent to a trace σ' of the specification (each local tester makes the same observation in σ and σ' despite σ and σ' differing). An example of this is given in Figure 5: in both scenarios the tester at port 1 observes $?i_1!o_1?i_1!o_1$ and the tester at port 2 observes $!o_2$.

Sometimes the local testers can be synchronised by the exchange of coordination messages[22, 23, 24]. In such situations, when testing from an FSM it is possible to overcome controllability and observability problems. For example, if input $?i_k$ at p is to be followed by input $?i_{k+1}$ at port $q \neq p$ and there is a controllability problem then this can be overcome by the tester at p sending a message to the tester at q after it sends input $?i_k$: this tells the tester at q that it can now supply its input. It is also possible to overcome observability problems when testing from an FSM. This observation has led to interest in two optimisation problem (when testing from an FSM): minimising the number of coordination messages used [25, 26]; and also minimising the number of channels between testers that are required [27, 28]. Unfortunately, however, there are some disadvantages associated with the use of coordination messages. First, their use leads to testing taking longer and requiring an additional network infrastructure. If there are timing constraints then it may not be possible for a tester to wait for a

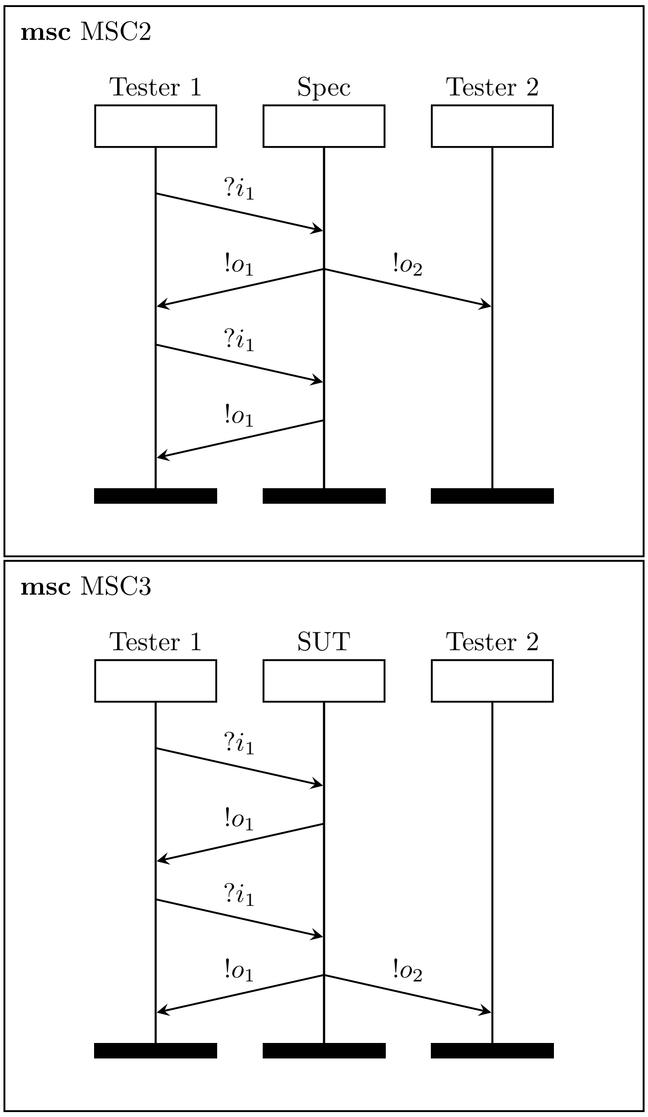

FIGURE 5. Observationally equivalent scenarios

coordination message and still satisfy these constraints. Second, if the coordination messages are sent using a network that is utilised by the SUT then this might change the network load and thus the behaviour of the SUT.

Much of the focus of work in distributed testing was on producing test sequences (from an FSM specification) that are controllable and also potentially that do not suffer from observability problems (see, for example, [24, 29, 30, 31, 32, 33]). However, the nature of observation in distributed testing leads to a different notion of correctness and this was reflected in the definition of an implementation relation for FSMs (when testing is controllable) [7] and later the implementation relation **dioco** for IOTSs [16]. Some recent work has explored test generation for these implementation relations [34, 16].

The use of these different implementation relations changes the nature of the Oracle problem. When testing from an FSM or IOTS, if testing is not distributed then the Oracle problem is that of determining whether a given sequence σ of inputs and outputs is also a trace of the specification and this is a classic (language) membership problem. In contrast, under **dioco** it is sufficient that the trace of the SUT is observationally equivalent to a trace of the specification. It has been shown that, as a result, the Oracle problem is NP-complete when applying distributed testing from an FSM [35]. Other work has looked at distributed testing from an IOTS and defined the implementation relation **mioco** [15]. However, this assumes that there is a global tester that observes all of the events; **mioco** differs from the classical IOTS implementation relation **ioco** by allowing the SUT to block all inputs on a channel. The situation in which the agent at port p only observes events at p has also been investigated for refinement in CSP [36].

Most work has focussed on testing from an FSM or an IOTS and here a behaviour is a sequence of events and concurrency is modelled using synchronisation and interleaving. There has also been some work that includes true concurrency in the model and this includes work that uses Partial Order Automata. Here, a transition is labelled by a partially ordered multi-set of inputs and outputs [37, 38]. There has also been interest in distributed testing from Petri Nets [39]. The main contribution of this line of work is that the models include true concurrency and, as a result, can be much more compact. For example, if we have three events a_1, a_2, and a_3 and these can occur in any order then an IOTS model would have to include all permutations while a Partial Order Automaton could include one transition with these events. For this example, using **ioco** leads to a combinatorial explosion but **dioco** need not since it is sufficient to model only one of the possible permutations, with the use of **dioco** implying that the others are allowed (since they are observationally equivalent). This paper considers the problem of testing from an IOTS model but it would be interesting to extend the work to models that allow true concurrency.

The work presented in this paper differs from previous work as follows. First, in the context of testing from an IOTS, this paper is the first to explore the notion of a tuple of local traces being an observation and to define an implementation relation in terms of this. Note, however, that one other piece of work independently considered the case where an observation is a tuple of (timed) local traces and investigated timed testing from timed models [21]. This other piece of work [21] assumed that the SUT and specification both have a component at each port and that we can observe messages sent between these components. Thus, the implementation relation **dtioco** defined requires there to be this structural similarity and also for internal events (message exchange) to be observed. It is thus very different from both **dioco** and

dioco$_o$. Importantly, we presented the first results that formally compare **dioco** with the alternative approach encapsulated in **dioco**$_o$ and thus to show that **dioco** is too strong when processes might be output-divergent. The results regarding the complexity of the Oracle problem are new but are related to previous results. An earlier piece of work proved that the Oracle problem is NP-hard for distributed testing but this concerned testing from an FSM [35]. FSMs differ significantly from IOTSs since, for example, input and output alternate and an FSM is quiescent after an output has been produced. A similar result has been shown for the corresponding problem when testing using **dioco** and IOTSs but is restricted to the case where the IOTSs are not output-divergent and also where the observation is a quiescent trace and not a tuple of local traces [44].

7. CONCLUSIONS

If testing involves physically distributed testers interacting with the SUT and these testers do not synchronised then we have distributed testing. Despite the growing importance of distributed systems, only relatively recently has an implementation relation **dioco** been defined for distributed testing. This paper explored the problem of determining whether an observation made in testing is allowed under **dioco**, with this being an instance of the Oracle problem.

In distributed testing each local tester observes a local trace and so an observation is a tuple $(\sigma_1, \ldots, \sigma_m)$ of local traces. The Oracle problem is thus that of deciding whether an observation $obs = (\sigma_1, \ldots, \sigma_m)$ is allowed by the specification s. This defines a verdict function v_s that maps $(\sigma_1, \ldots, \sigma_m)$ to $pass$ if s has a trace σ such that each σ_p is a prefix of the projection $\pi_p(\sigma)$ of σ onto p and otherwise the observation is mapped to $fail$. We proved that this verdict function is both sound and exhaustive for **dioco** if processes are not output-divergent: r **dioco** s if and only if all observations that can be made when testing r are mapped to $pass$. Thus, for processes that are not output-divergent we have that **dioco** is equivalent to the implementation relation **dioco**$_o$ implied by the verdict function.

We found that if we consider output-divergent processes then **dioco** is strictly stronger than **dioco**$_o$. In addition, there are cases where an IOTS r does not conform to IOTS s under **dioco** but this cannot be determined based on finite observations. This suggests that the definition of **dioco** is too strong for testing when processes might be output-divergent and **dioco**$_o$ may be more suitable. We then proved that **dioco** and **dioco**$_o$ are identical for output-divergent processes if we restrict attention to processes that are finitely-branching and do not allow starvation. Finally, we considered the problem of computing the verdict function and proved that this is NP-complete but can be solved in polynomial time if we have an upper bound on the number of ports.

There are several lines of future work. First, there may be additional interesting conditions under which the Oracle problem can be solved in polynomial time. There may also be additional sensible classes of IOTSs that are output-divergent but where **dioco** and **dioco**$_o$ converge. It would also be interesting to explore what happens if we add features such as time and probabilities. Finally, there is the issue of considering true concurrency models.

REFERENCES

[1] Farchi, E., Hartman, A., and Pinter, S. S. (2002) Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, **41**, 89–110.

[2] Grieskamp, W., Kicillof, N., Stobie, K., and Braberman, V. (2011) Model-based quality assurance of protocol documentation: tools and methodology. *The Journal of Software Testing, Verification and Reliability*, **21**, 55–71.

[3] Chow, T. S. (1978) Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, **4**, 178–187.

[4] Dssouli, R. and von Bochmann, G. (1985) Error detection with multiple observers. *Protocol Specification, Testing and Verification V*, pp. 483–494. Elsevier Science (North Holland).

[5] Dssouli, R. and von Bochmann, G. (1986) Conformance testing with multiple observers. *Protocol Specification, Testing and Verification VI*, pp. 217–229. Elsevier Science (North Holland).

[6] Hennie, F. C. (1964) Fault-detecting experiments for sequential circuits. *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, Princeton, New Jersey, November, pp. 95–110.

[7] Hierons, R. M. and Ural, H. (2008) The effect of the distributed test architecture on the power of testing. *The Computer Journal*, **51**, 497–510.

[8] Moore, E. F. (1956) Gedanken-experiments. In Shannon, C. and McCarthy, J. (eds.), *Automata Studies*. Princeton University Press.

[9] Petrenko, A., Yevtushenko, N., Lebedev, A., and Das, A. (1994) Nondeterministic state machines in protocol conformance testing. *Proceedings of Protocol Test Systems, VI (C-19)*, Pau, France, 28-30 September, pp. 363–378. Elsevier Science (North-Holland).

[10] Petrenko, A., Yevtushenko, N., and v. Bochmann, G. (1996) Testing deterministic implementations from nondeterministic FSM specifications. *Testing of Communicating Systems, IFIP TC6 9th International Workshop on Testing of Communicating Systems*, Darmstadt, Germany, 9–11 September, pp. 125–141. Chapman and Hall.

[11] Petrenko, A. and Yevtushenko, N. (2005) Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers*, **54**, 1154–1165.

[12] Sarikaya, B. and Bochmann, G. (1984) Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications*, **32**, 389–395.

[13] Yevtushenko, N. V., Lebedev, A. V., and Petrenko, A. F. (1991) On checking experiments with nondeter-

ministic automata. *Automatic Control and Computer Sciences*, **6**, 81–85.

[14] Brinksma, E. (1988) A theory for the derivation of tests. *Proceedings of Protocol Specification, Testing, and Verification VIII*, Atlantic City, pp. 63–74. North-Holland.

[15] Brinksma, E., Heerink, L., and Tretmans, J. (1998) Factorized test generation for multi-input/output transition systems. *11th IFIP International Workshop on Testing Communicating Systems (IWTCS)*, IFIP Conference Proceedings, **131**, pp. 67–82. Kluwer.

[16] Hierons, R. M., Merayo, M. G., and Núñez, M. (2012) Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, **25**, 35–62.

[17] Tretmans, J. (1996) Conformance testing with labelled transitions systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, **29**, 49–79.

[18] Tretmans, J. (2008) Model based testing with labelled transition systems. *Formal Methods and Testing*, Lecture Notes in Computer Science, **4949**, pp. 1–38. Springer.

[19] Joint Technical Committee ISO/IEC JTC 1 (1994) *International Standard ISO/IEC 9646-1. Information Technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts.* ISO/IEC.

[20] Hierons, R. M., Merayo, M. G., and Núñez, M. (2008) Implementation relations for the distributed test architecture. *20th IFIP TC 6/WG 6.1 International Conference on the Testing of Software and Communicating Systems (TestCom/FATES 2008)*, Lecture Notes in Computer Science, **5047**, pp. 200–215. Springer.

[21] Gaston, C., Hierons, R. M., and Gall, P. L. (2013) An implementation relation and test framework for timed distributed systems. *25th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2013)*, Lecture Notes in Computer Science, **8254**, pp. 82–97. Springer.

[22] de Almeida, E. C., Marynowski, J. E., Sunyé, G., Traon, Y. L., and Valduriez, P. (2010) Efficient distributed test architectures for large-scale systems. *22nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2010)*, Lecture Notes in Computer Science, **6435**, pp. 174–187. Springer.

[23] Rafiq, O. and Cacciari, L. (2003) Coordination algorithm for distributed testing. *The Journal of Supercomputing*, **24**, 203–211.

[24] Chen, W.-H. and Ural, H. (1995) Synchronizable checking sequences based on multiple UIO sequences. *IEEE/ACM Transactions on Networking*, **3**, 152–157.

[25] Hierons, R. M. (2001) Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults. *Information and Software Technology*, **43**, 551–560.

[26] Wu, W.-J., Chen, W.-H., and Tang, C. Y. (1998) Synchronizable test sequence for multi-party protocol conformance testing. *Computer Communications*, **21**, 1177–1183.

[27] Jourdan, G.-V., Ural, H., and Yenigün, H. (2006) Minimizing coordination channels in distributed testing. *Formal Techniques for Networked and Distributed Systems (FORTE 2006)*, Lecture Notes in Computer Science, **4229**, pp. 451–466. Springer.

[28] Hierons, R. M. and Ural, H. (2009) Overcoming controllability problems with fewest channels between testers. *Computer Networks*, **53**, 680–690.

[29] Hierons, R. M. and Ural, H. (2003) UIO sequence based checking sequences for distributed test architectures. *Information and Software Technology*, **45**, 793–803.

[30] Hierons, R. M. and Ural, H. (2008) Checking sequences for distributed test architectures. *Distributed Computing*, **21**, 223–238.

[31] Luo, G., Dssouli, R., and v. Bochmann, G. (1993) Generating synchronizable test sequences based on finite state machine with distributed ports. *The 6th IFIP Workshop on Protocol Test Systems*, pp. 139–153. Elsevier (North-Holland).

[32] Tai, K.-C. and Young, Y.-C. (1998) Synchronizable test sequences of finite state machines. *Computer Networks and ISDN Systems*, **30**, 1111–1134.

[33] Young, Y. C. and Tai, K. C. (1998) Observational inaccuracy in conformance testing with multiple testers. *IEEE 1st workshop on application-specific software engineering and technology*, pp. 80–85.

[34] Hierons, R. M. Generating complete controllable test suites for distributed testing. *IEEE Transactions on Software Engineering*, **to appear**.

[35] Hierons, R. M. (2012) Oracles for distributed testing. *IEEE Transactions on Software Engineering*, **38**, 629–641.

[36] Jacob, J. L. (1989) Refinement of shared systems. In McDermid, J. (ed.), *The Theory and Practice of Refinement: Approaches to the Formal Development of Large-Scale Software Systems*, pp. 27–36. Butterworths.

[37] Haar, S., Jard, C., and Jourdan, G.-V. (2007) Testing input/output partial order automata. *19th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems (TestCom/FATES)*, Lecture Notes in Computer Science, **4581**, pp. 171–185. Springer.

[38] von Bochmann, G., Haar, S., Jard, C., and Jourdan, G.-V. (2008) Testing systems specified as partial order input/output automata. *20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems (TestCom/FATES)*, Lecture Notes in Computer Science, **5047**, pp. 169–183. Springer.

[39] de León, H. P., Haar, S., and Longuet, D. (2013) Unfolding-based test selection for concurrent conformance. *25th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2013)*, Lecture Notes in Computer Science, **8254**, pp. 98–113. Springer.

[40] Hierons, R. M. (2009) Verdict functions in testing with a fault domain or test hypotheses. *ACM Transactions on Software Engineering and Methodology*, **18**.

[41] Lynch, N. and Vaandrager, F. (1991) Forward and backward simulations for timing-based systems. *REX Workshop "Real-Time: Theory in Practice", LNCS 600*, pp. 397–446. Springer.

[42] Schaefer, T. J. (1978) The complexity of satisfiability problems. *Tenth Annual ACM Symposium on Theory of Computing (STOC)*, pp. 216–226.

[43] Kozen, D. (1977) Lower bounds for natural proof systems. *FOCS*, pp. 254–266.

[44] Hierons, R. M. (2014) Combining centralised and distributed testing. *ACM Transactions on Software Engineering and Methodology*, **24**, 5:1–5:29.