Combining centralised and distributed testing

Robert M. Hierons, Brunel University

Many systems interact with their environment at distributed interfaces (ports) and sometimes it is not possible to place synchronised local testers at the ports of the system under test (SUT). There are then two main approaches to testing: having independent local testers or a single centralised tester that interacts asynchronously with the SUT. The power of using independent testers has been captured using implementation relation **dioco**. In this paper we define implementation relation $\textbf{dioco}_c$ for the centralised approach and prove that **dioco** and $\textbf{dioco}_c$ are incomparable. This shows that the frameworks detect different types of faults and so we devise a hybrid framework and define an implementation relation $\textbf{dioco}_s$ for this. We prove that the hybrid framework is more powerful than the distributed and centralised approaches. We then prove that the Oracle problem is NP-complete for $\textbf{dioco}_c$ and $\textbf{dioco}_s$ but can be solved in polynomial time if we place an upper bound on the number of ports. Finally, we consider the problem of deciding whether there is a test case that is guaranteed to force a finite state model into a particular state or to distinguish two states, proving that both problems are undecidable for the centralised and hybrid frameworks.

Categories and Subject Descriptors: B4.5 [**Reliability, Testing, and Fault-Tolerance**]: Test generation; C.1.4 [**Parallel Architectures**]: Distributed architectures; D2.5 [**Testing and Debugging**]: Testing Tools; H.3.4 [**Systems and Software**]: Distributed systems

General Terms: Reliability, Verification

Additional Key Words and Phrases: Centralised testing, distributed testing, model based testing

ACM Reference Format:
R. M. Hierons, 2014. Combining centralised and distributed testing. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 29 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Testing is a significant part of the software development process but is typically manual and, as a result, expensive and error prone. This observation has led to interest in automating testing including work on model-based testing (MBT) in which testing is based on a model of the required behaviour of the *system under test (SUT)* or some aspect of this (see, for example, [En-Nouaary 2013; Farchi et al. 2002; Grieskamp 2006; Hwang et al. 2012; Miller and Strooper 2012; Tahat et al. 2012; Tretmans 1996]). Given a model, there is the potential to automate test case generation and execution but also to automatically check that an observed behaviour is one allowed: the model acts as an Oracle. Recent industrial experience has shown that the use of MBT can lead to significant reductions in the cost of testing [Grieskamp et al. 2011] but MBT builds on much earlier work in the context of automata theory (see, for example, [Moore 1956; Hennie 1964; Chow 1978]).

Author's addresses: R.M. Hierons, Department of Computer Science, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK

Most work in MBT has focussed on testing from either finite state machines (FSMs) (see, for example, [Moore 1956; Hennie 1964; Gonenc 1970; Chow 1978; Lee and Yannakakis 1996; Hierons and Ural 2008b]) or input output transition systems (IOTSs) (see, for example, [Tretmans 1996]). While the tester might produce a model in a more expressive language, such models are typically mapped to an FSM or IOTS by the MBT tool used (see, for example, [Farchi et al. 2002; Grieskamp 2006]). Typically, a model in a language such as state-charts can be mapped to an FSM or IOTS by either abstracting out the data or expanding the data. In the former the data is simply ignored, something that is possible when the data is not referred to in guards of transitions and so does not affect whether a path through a model is feasible. In the latter approach, each combination of logical state (of the model) and tuple of values of the model variables potentially forms a separate state of the IOTS or FSM (see, for example, [Petrenko et al. 2004]). A model written using a process algebra such as LOTOS can also be given an IOTS semantics. [Tretmans 2008].

The classic notion of correctness (implementation relation) used with IOTSs is **ioco** [Tretmans 1996]. Recent work has showed that **ioco** is equivalent to an alternative implementation relation called alternating simulation, used with interface automata, if the IOTSs are input-enabled[1] [Veanes and Bjørner 2010]. As a result, it should be possible to adapt methods devised for IOTSs and **ioco** to languages whose semantics is expressed using alternating simulation. It has also been shown that, by including the notion of inputs being illegal in some states, it is possible to map between interface automata and FSMs [Aarts and Vaandrager 2010].

When testing from a model M, we need to say what it means for the SUT to be correct. In this context, it is normal to assume that the SUT behaves like some unknown model N, typically written using the same formalism as M. This assumption is sometimes called the minimum hypothesis [Gaudel 1995] and allows one to formally state what it means for the SUT to be a correct implementation of M by defining an *implementation relation* between models N and M. It is important to use a suitable implementation relation since this can be involved in driving test generation (the implementation relation states what behaviours constitute failures) and is used in solving the Oracle problem (of checking an observation against the specification). If we use the wrong implementation relation then automated testing might use inappropriate test cases (ones that cannot lead to failures) and might also give the wrong outcome (pass/fail) for a test.

Many systems interact with their environment at physically distributed interfaces, called *ports*, with important classes of such systems including communications protocols, web services, cloud systems, and wireless sensor networks. In testing such a system, it is sometimes possible to place separate testers at the ports and coordinate testing through the testers exchanging messages or being controlled by a central coordinator via message exchange [Cacciari and Rafiq 1999; Jard et al. 1998; Rafiq and Cacciari 2003]. However, this message exchange introduces a network overhead and can also introduce delays in testing since message exchange is not instantaneous. Two alternatives, that do not require additional coordination messages, have been described in the literature:

— Place a separate independent tester at each interface of the SUT and require these testers to act independently in communicating with the SUT [Chen et al. 2005; Chen and Ural 1995; Dssouli and von Bochmann 1985; 1986; Hierons and Ural 2008a].

[1]An IOTS is input-enabled if for every state s and input $?i$ there is at least one transition from s with label $?i$.

The tester at a port p has its own strategy, only observes events that occur at p, and interacts synchronously with the port p of the SUT.
— Have a single centralised tester that interacts asynchronously with the separate ports of the SUT [Simão and Petrenko 2011].

Recent work has shown that the use of separate independent testers significantly affects testing. For example, even when testing from a deterministic finite state machine M, it is undecidable whether there is a test case that is guaranteed to reach a particular state s of M or distinguish two given states of M [Hierons 2010]. In addition, the Oracle problem (of deciding whether an observation is consistent with the specification) is NP-complete [Hierons 2012b]. This contrasts with the case in which there is only a single port, where it is decidable whether there is a test case that reaches a state or distinguishes two states [Alur et al. 1995] and the Oracle problem corresponds to the membership problem for finite automata and so can be solved in low-order polynomial time. One of the reasons for the increased complexity of testing is that the testers cannot coordinate their actions and so each tester has only partial information about the sequence of events that has occurred. The alternative, of having a centralised tester, has the advantage of having only one tester and so no need to coordinate testing. However, it has the disadvantage that communication with the SUT is asynchronous and so there can be a loss of information regarding the relative order of inputs and outputs. For example, if the tester sends input $?i$, then sends input $?i'$, and finally observes output $!o$ then while it might appear that $!o$ was produced in response to $?i'$, it is possible that $!o$ was output before $?i'$ was received by the SUT and the observed order of events resulted from the communications latency. Interestingly, it appears that the centralised and distributed approaches have not previously been formally compared. Since the centralised approach has been described for input output transition systems (IOTSs) and the distributed approach has been investigated for both finite state machines and IOTSs, this paper compares these approaches in the context of testing from an IOTS (the more general formalism).

We show that the centralised and distributed approaches are incomparable and so they will identify different sets of *traces* (sequences of inputs and outputs) of an SUT that are not defined by the specification. We therefore define a hybrid framework that combines these two approaches by having a centralised tester that interacts asynchronously with the SUT and local testers that observe local traces. We prove that this framework is strictly more effective than either the centralised or distributed approaches. Having defined the hybrid framework we investigate some standard testing problems. We prove that the Oracle problem is NP-complete for the hybrid and centralised approaches (it is already known that this is NP-complete for distributed testing [Hierons 2012b]). We also give an algorithm for the Oracle problem that operates in polynomial time when there is an upper bound on the number of ports. Thus, it appears that the Oracle problem being NP-complete may not be problematic for systems where there are only a few ports, an example being communications protocols where there are two ports. We then consider the problems of deciding whether there is a test case that is guaranteed to force an IOTS M into a given state s and whether there is a test case that is guaranteed to distinguish two states s and s' of an IOTS. We prove that these problems are generally undecidable for both the centralised and hybrid approaches.

In practice, one might allow the testers to exchange coordination messages in order to support testing and this topic has received much attention (see, for example, [Chen and Ural 1995; Cacciari and Rafiq 1999; Jourdan et al. 2006; Tai and Young 1998]). However, when testing a system with physically distributed interfaces (ports), the exchange of coordination messages is asynchronous and so this approach might add no value beyond the centralised framework[2] and it will then be weaker than the hybrid framework described in this paper. The problems addressed are therefore inherent in testing a system that has physically distributed interfaces: either we have independent distributed testers that interact synchronously with the SUT or we have one or more testers that interact asynchronously with the SUT. The use of coordination messages does not solve the underlying problem since the exchange of such coordination messages will be asynchronous.

The work in this paper relates to several lines of research. The early work in the area of distributed testing focussed on testing from an FSM and this work demonstrated that having distributed observations affects the ability of testing to distinguish between an SUT and a specification [Sarikaya and von Bochmann 1984; Dssouli and von Bochmann 1985; 1986]. This work also showed that the use of distributed testers can introduce controllability problems, which can make it harder to apply a given test case; essentially, the tester at a port p might not know when to apply an input since it does not observe previous events at the other ports. Another line of work defined an implementation relation for distributed testing from an FSM for the case where we restrict attention to test cases that introduce no controllability problems [Hierons and Ural 2008b]. Only relatively recently have researchers investigated distributed testing when the specification is an IOTS, defining the implementation relation **dioco** for the case where there is a separate tester at each port of the SUT [Hierons et al. 2012]. It has also been shown that the Oracle problem is NP-complete for **dioco** [Hierons 2012b]. Earlier work defined the implementation relation **mioco** that adapts the implementation relation **ioco**, typically used for testing from a single port IOTS, for testing distributed systems. However, **mioco** does not take into account the distributed nature of observations: observations are still global traces with **mioco** differing from **ioco** through the SUT being allowed to block all input at a port (see, for example, [Brinksma et al. 1998; Heerink and Tretmans 1997; Li et al. 2004]). Recent work has investigated testing when interacting with the SUT through asynchronous channels [Hierons 2012a; Noroozi et al. 2011; Simão and Petrenko 2011; Weiglhofer and Wotawa 2009], but has only considered single-port systems. This work has shown that for first-in-first-out (FIFO) channels, it is possible to decide whether there is a test case that is guaranteed to move an IOTS into a particular state as long as the specification is not output-divergent[3] [Hierons 2012a]. It has also defined implementation relations and shown that it is generally undecidable whether a model N of the SUT conforms to a specification M [Hierons 2013]. This latter result, that conformance is undecidable, immediately extends to centralised testing of an SUT that has distributed interfaces since the above result is for the special case where the SUT has only one port. There has also been work that considers the case where a single centralised tester interacts asynchronously with the separate ports of the SUT, an approach that the hybrid framework extends [Simão and Petrenko 2011].

There are several practical factors that motivate the work in this paper. First, many systems interact with their environment at multiple physically distributed ports with examples including communications protocols, web services, cloud systems, and wireless sensor networks. The problem of testing such systems is therefore important and, as noted above, it is not always feasible or desirable to synchronise testing through the exchange of coordination messages. It is important to use the correct implemen-

[2]It is, however, known that testing can be synchronised using coordination messages when testing from a deterministic finite state machine since input and output alternate: having produced an output the SUT waits for the next input.

[3]An IOTS is output-divergent if it has a state from which it can take an infinite path that does not contain inputs.

tation relation since this relation drives testing (it states what behaviours would constitute failures and so what test cases are potentially useful) and is also the basis of an automated Oracle. As a result, if we use the wrong implementation relation then testing can be inefficient and/or unsound. Having defined implementation relations it is also possible to formally reason about the differences between an SUT and a specification that can be found by testing and by comparing implementation relations we identify weaknesses in the previously discussed distributed and centralised approaches. We define a stronger approach, the hybrid framework, which is strictly more effective (in finding differences between the SUT and specification) than the centralised and distributed approaches. Interestingly, it transpires that the hybrid framework is more effective than separately applying both the centralised and distributed approaches. The decision problems examined in this paper are also important in the context of test automation: they either correspond to checking an observation against a model/specification (the Oracle problem) or to finding test cases that achieve certain objectives (reaching a state or distinguishing two states). The complexity and decidability results have the potential to inform practice and lead to notions of testability for distributed testing.

As previously discussed, the initial work regarding distributed testing was carried out in the context of communications protocols. Here there are two testers: when testing the implementation N of a protocol there is an upper tester U that acts as the layer above N (it attempts to communicate using features provided by N) and there is a lower tester L that sits on a separate machine. The use of separate independent testers for protocol conformance testing was formalised by ISO as the distributed test architecture [ISO/IEC 1995]. Now there are, however, many other classes of systems that interact with their environment at physically distributed interfaces. Consider, for example, the networked controllers in a car, aircraft or a manufacturing plant. These controllers access information from different sensors in real time, leading to interest in decentralised control (see, for example [Yang et al. 1999; Swigart and Lall 2011]) and potentially the need for distributed testing. While it may be possible to test individual components by employing a single tester, it seems likely that the testing of the system as a whole will require a distributed approach. Many companies have web services, with some functions potentially requiring the involvement of customers and staff at different locations, and here testing is likely to be distributed. There has also been increasing interest in online gaming; again, there are customers and staff at physically distributed locations. It seems likely that the importance of distribution will only increase if organisations continue to move some services to cloud systems.

The following are the main contributions of the paper. First, we define an implementation relation $\mathbf{dioco}_c$ that corresponds to centralised testing of an SUT through asynchronous message exchange. We then prove that the implementation relations **dioco** and $\mathbf{dioco}_c$ are incomparable and so the corresponding approaches to testing find different classes of differences between the SUT and the specification. This leads to the definition of a hybrid framework and a corresponding implementation relation $\mathbf{dioco}_s$ that is strictly stronger than both **dioco** and $\mathbf{dioco}_c$. We therefore know that this hybrid framework is more effective than the two previously described approaches. We then prove that the Oracle problem is NP-complete for both $\mathbf{dioco}_c$ and $\mathbf{dioco}_s$ but can be solved in polynomial time if there is an upper bound on the number of ports. We also show that, for the hybrid and centralised frameworks, it is undecidable whether there is a test case that is guaranteed to take an IOTS to a particular state or to distinguish two given states. The proof uses an IOTS that can be represented using an FSM and so the result also holds for FSMs.

The paper is structured as follows. In Section 2 we start by defining IOTSs and associated notation. In Section 3 we define an implementation relation $\mathbf{dioco}_c$ for cen-

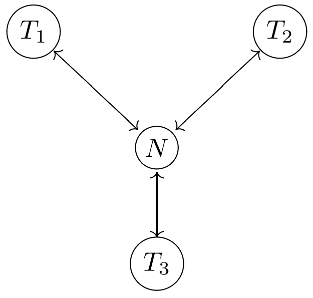

Fig. 1. Distributed Test Architecture

tralised testing of an SUT with distributed ports. In Section 4 we prove that **dioco** and **dioco**$_c$ are incomparable. We therefore define a hybrid framework and corresponding implementation relation **dioco**$_s$ and prove that **dioco**$_s$ is strictly stronger than both **dioco** and **dioco**$_c$. Section 5 then explores the Oracle problem, proving that it is NP-complete for both **dioco**$_c$ and **dioco**$_s$. In Section 6, we prove that, when applying a centralised or hybrid approach, it is undecidable whether there is a test case that is guaranteed to reach a particular state or distinguish two given states. Finally, we draw conclusions and discuss future work in Section 7.

2. PRELIMINARIES

2.1. Observations

This paper considers systems that interact with their environment at $n > 1$ physically distributed ports and we let $\mathcal{P} = \{1, \ldots, n\}$ denote the set of names of the ports. We let I denote the set of inputs that the SUT might receive and O denote the set of outputs that it might potentially produce. Given port p we let I_p denote the set of inputs at p and O_p denote the set of outputs at p. We assume that the I_p partition I and the O_p partition O: if necessary we can label inputs and outputs with the corresponding port number to ensure that this is the case. We use the normal convention where the name of an input is preceded by ? and the name of an output is preceded by !. We use subscripts to denote the port number and so, for example, $?i_1$ denotes an input at port 1 and $!o_2$ denotes an output at port 2.

The distributed test architecture is illustrated by Figure 1 in which T_1, T_2, and T_3 are the local testers and N is the SUT. The essential idea is that each tester interacts synchronously with the SUT and only observes the events (inputs and outputs) involved in its interaction with the SUT. Thus, no tester observes the global trace produced: each observes the projection in which it is involved, with this observation being called a *local trace*. Let us suppose, for example, that the SUT produces a trace $?i_1?i_2!o_2$ where input $?i_1$ is an input at port 1, $?i_2$ is an input at port 2, and $!o_2$ is an output at port 2. In this case the SUT produced $?i_1?i_2!o_2$, the tester at port 1 observed $?i_1$, and the tester at port 2 observed $?i_2!o_2$. The testers are called *local testers* in order differentiate this situation from the case where there is a single global tester that observes all of the events.

Work in the context of testing from an IOTS often considers quiescence: the SUT is quiescent if it cannot change state or produce output without first receiving input. It is often assumed that quiescence can be observed and δ is used to denote the observation of quiescence (in this paper we consider both the case where quiescent is observable and where it is not). Throughout the paper we let $\mathcal{A}ct = I \cup O \cup \{\delta\}$ denote the set of possible events and given port $p \in \mathcal{P}$ we let $\mathcal{A}ct_p = I_p \cup O_p \cup \{\delta\}$ denote the set of events that can be observed at p.

As usual, given a set A we use A^* to denote the set of finite sequences of elements of A. When we have separate testers at the ports of the SUT, the tester at port p observes only the events at p. Thus, if the SUT produces (global) trace σ then the tester at $p \in \mathcal{P}$ observes the local trace $\pi_p(\sigma)$ defined by the following in which $\sigma \in \mathcal{A}ct^*$ and $a \in \mathcal{A}ct$ (see, for example, [Hierons and Ural 2008b]).

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

Two traces σ_1 and σ_2 cannot be distinguished when using local testers if they have the same sets of corresponding local traces and we then write $\sigma_1 \sim \sigma_2$. More formally, $\sigma_1 \sim \sigma_2$ if for all $p \in \mathcal{P}$ we have that $\pi_p(\sigma_1) = \pi_p(\sigma_2)$.

For centralised testing we assume that there are asynchronous channels between the tester and the ports of the SUT. We assume that these channels are first-in-first-out (FIFO); the use of non-FIFO channels is a topic for future work and will be discussed further in Section 7.

2.2. Input output transition systems

We start by defining input output transition systems.

Definition 2.1. An input output transition system (IOTS) M is defined by a tuple (S, I, O, T, s_0) in which S is the countable set of states, $s_0 \in S$ is the initial state, I is the countable set of inputs, O is the countable set of outputs, and $T \subseteq S \times (I \cup O \cup \{\tau\}) \times S$, where τ represents internal (unobservable) events, is the transition relation. A transition (s, a, s') should be interpreted as meaning that from state s it is possible to move to state s' with event $a \in I \cup O \cup \{\tau\}$. We assume that I and O are disjoint and $\tau \notin I \cup O$. State $s \in S$ is said to be *quiescent* if from s it is not possible to change state or produce output without first receiving input and δ is used to represent the tester observing quiescence. We can extend T, the transition relation, to T_δ by adding the transition (s, δ, s) for each quiescent state s. IOTS M is *input enabled* if for all $s \in S$ and $?i \in I$ there exists $s' \in S$ such that $(s, ?i, s') \in T$. An IOTS is *output-divergent* if it can reach a state in which there is an infinite path that contains outputs and internal events only.

Figure 2 gives an IOTS from [Hierons et al. 2012] that represents a distributed majority voting system; we will call this M_0. Two agents U and L interact with a system via their terminals. The initial state of M_0 is s_0; this is shown twice in order to simplify the diagram. The system starts by sending output $!r_U$ to port U and then $!r_L$ to port L to tell the agents that a poll is to start. Each agent then replies with a vote: $?l_0$ and $?l_1$ denote agent l voting 0 and 1 respectively and, similarly, $?u_0$ and $?u_1$ denote agent U voting 0 and 1 respectively. If the votes are identical then the SUT sends confirmation to the two agents; 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 $?x$ there is an implicit self-loop transition from s to s with input $?x$.

The global traces of IOTS M_0 include $\sigma = !r_U!r_L?l_1?u_1!1_U!1_L$ and this has corresponding local traces $\pi_L(\sigma) = !r_L?l_1!1_L$ and $\pi_U(\sigma) = !r_U?u_1!1_U$. It is straightforward to see that M_0 is not output-divergent but that not all of its states are quiescent. For example, s_0

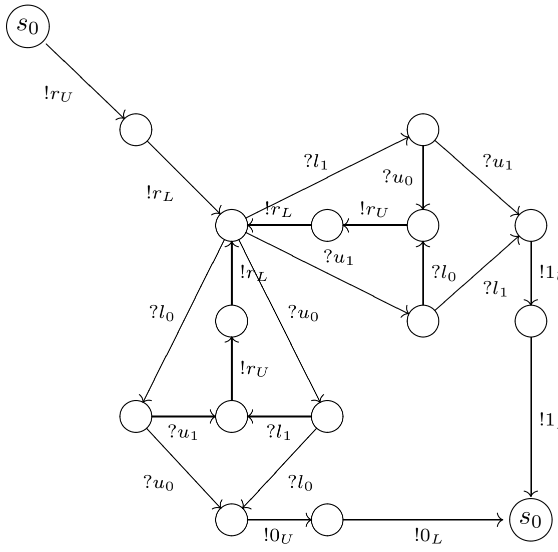

Fig. 2. IOTS M_0

is not quiescent since there is a transition from s_0 with output $!r_U$. In contrast, the state reached from s_0 by $!r_U!r_L$ is quiescent.

Let us suppose that we wish to test an SUT that should implement M_0 and we are interested in testing the behaviour that corresponds to trace $\sigma = !r_U!r_L?l_1?u_1!1_U!1_L$. In distributed testing there would be a single tester at each port. The tester at U would wait to observe output $!r_U$, would apply $?u_1$, and then wait to observe $!1_U$. Similarly, the tester at L would wait to observe output $!r_L$, would apply $?l_1$, and then wait to observe $!1_L$. If, for example, the SUT produced trace $\sigma' = !r_U?u_1!r_L?l_1!1_U!1_L$ then the local testers would observe the expected local traces $!r_U?u_1!1_U$ and $!r_L?l_1!1_L$ and so σ' would not lead to a failure. In the centralised approach the central tester might wait to observe $!r_U$ and $!r_L$ (in either order), then send $?u_1$ and $?l_1$ and finally wait to observe $!1_U$ and $!1_L$. Here, if the tester produced trace $\sigma'' = !r_U!r_L!1_U!1_L?l_1?u_1$ then the tester might still observe σ due to the message latency: even though the SUT produces $!1_U$ and $!1_L$ before receiving $?l_1$ and $?u_1$ the message delay leads to the tester observing $?l_1$ and $?u_1$ before $!1_U$ and $!1_L$.

As usual, in this paper we assume that all processes are input-enabled and are not output divergent. If a process is not input-enabled then typically it is possible to complete this process by adding either self-loops (denoting that an unspecified input should have no effect) or transitions to an error state (denoting the situation in which the SUT is allowed to do anything after an unspecified input). While this is not always possible, since an input $?i$ not being specified in state s might correspond to the situation where $?i$ should not be received in state s, only considering input-enabled processes is not a significant restriction. We will restrict attention to processes that are not output-divergent since, as we will see, we wish to only consider traces of the SUT that take it to a quiescent state; as long as the SUT is not output-divergent, every trace of the SUT is a prefix of such a trace. In addition, output-divergence can be seen as being similar to a livelock and might reflect a failure. An alternative devised for **dioco**[4] is to define

[4]Distributed input-output conformance.

the implementation relation in terms of infinite traces [Hierons et al. 2012] but this complicates the exposition.

Definition 2.2. Given IOTS $M = (S, I, O, T, s_0)$ we use the following notation.

(1) If $(s, a, s') \in T_\delta$, for $a \in \mathcal{A}ct \cup \{\tau\}$, then we write $s \xrightarrow{a} s'$.
(2) We write $s \overset{\epsilon}{\Longrightarrow} s'$ if there exist $s_1, \ldots, s_k$, for $k \geq 1$, such that $s = s_1$, $s' = s_k$, $s_1 \xrightarrow{\tau} s_2, \ldots, s_{k-1} \xrightarrow{\tau} s_k$.
(3) We write $s \overset{a}{\Longrightarrow} s'$, for $a \in \mathcal{A}ct$, if there exist s_1, s_2 such that $s \overset{\epsilon}{\Longrightarrow} s_1$, $s_1 \xrightarrow{a} s_2$, and $s_2 \overset{\epsilon}{\Longrightarrow} s'$.
(4) We write $s \overset{\sigma}{\Longrightarrow} s'$ for $\sigma = a_1 \ldots a_k \in \mathcal{A}ct^*$ if there exist $s_1, \ldots, s_{k+1}$, $s = s_1$, $s' = s_{k+1}$ such that for all $1 \leq i < k$ we have that $s_i \overset{a_i}{\Longrightarrow} s_{i+1}$.
(5) We write $s \overset{\sigma}{\Longrightarrow}$ if there exists s' such that $s \overset{\sigma}{\Longrightarrow} s'$ and we say that σ is a *trace* of M if $s_0 \overset{\sigma}{\Longrightarrow}$. We let $\mathcal{T}r(M)$ denote the set of traces of M. We say that trace σ of M is a *quiescent trace* if there is a quiescent state s of M such that $s_0 \overset{\sigma}{\Longrightarrow} s$.

Much of the research in testing state-based systems assumes that a test case is an input sequence. However, if we have nondeterminism in either the specification or the SUT then it can be useful to have adaptive test cases: test cases where the next action of the tester (send an input or wait and observe output) depends on the sequence of events that has been observed. Such test cases correspond to the notion of a strategy in game theory [Alur et al. 1995; Lee and Yannakakis 1994]. However, in the proof that it is undecidable whether there is a test case that is guaranteed to take an IOTS to a particular state we will use IOTSs with no outputs. Here adaptivity adds nothing and so it is sufficient to consider test cases that are input sequences and we do this since it simplifies the discussion.

3. IMPLEMENTATION RELATIONS FOR CENTRALISED AND DISTRIBUTED TESTING

The types of testing considered in this paper introduce implementation relations, which state whether testing can distinguish an SUT process N from a specification process M. An implementation relation captures the power of a particular type of testing: its ability to lead to differences between N and M being observed.

One important decision to make regarding testing is the role of quiescence. Much of the work on testing from an IOTS assumes that quiescence is observed and, as noted earlier, represents this observation using δ. However, when communicating with an SUT through asynchronous channels the observation of quiescence introduces additional issues since it requires either knowledge regarding message latency or the ability to determine the current contents of the channels. This may be feasible for some distributed systems that are on one site but otherwise it seems likely that this assumption will not hold. We therefore define two versions of each implementation relation: one in which we allow quiescence to be observed and one where we do not. However, we will assume that a complete test takes the SUT to a quiescent state since any output produced will eventually be observed. It will transpire that the main results in this paper are not affected by whether we can observe quiescence in testing: we prove the results for the case where quiescence is not observed but comment on what happens when we include quiescence.

First, we review the implementation relation **dioco** introduced for distributed testing from an IOTS. The **dioco** implementation relation is a version of the implementation relation **ioco** [Tretmans 1996], which is widely used in work on testing from an IOTS that has a single port.

When we have independent distributed testers, the tester at port p observes the projection $\pi_p(\sigma)$ of the global trace σ that occurs. After testing has finished the projections are brought together and compared with the traces of the specification. We use projections of traces ending in quiescence in order to ensure that the local observations are all projections of the same global trace. To see why this is the case, consider the situation where the specification is M_0 and the SUT produces output sequence $!r_U!r_L$ before becoming quiescent. If we do not restrict ourselves to making observations in quiescent states then the tester at port U might stop testing before it observes $!r_U$ and the tester at port L might still observe $!r_L$ since the testers do not synchronise. We might then incorrectly conclude that the SUT produced a trace starting with $!r_L$. This could lead to the verdict fail if the SUT produced $!r_U!r_L$, even though $!r_U!r_L$ is a trace of the specification M_0.

The observation of projections of quiescent traces leads to the following definition of implementation relation **dioco** [Hierons et al. 2012].

Definition 3.1. Given IOTSs N and M with the same sets of inputs and outputs we have that N **dioco** M if for every trace $\sigma \in \mathcal{A}ct^*$ such that $\sigma\delta \in \mathcal{T}r(N)$ we have that there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma' \sim \sigma$.

The following adapts **dioco** to the case where we do not observe quiescence during testing.

Definition 3.2. Given IOTSs N and M with the same sets of inputs and outputs we have that N **dioco**$'$ M if for every trace $\sigma \in (\mathcal{A}ct \setminus \{\delta\})^*$ such that $\sigma\delta \in \mathcal{T}r(N)$ we have that there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma' \sim \sigma$.

As noted earlier, since we only consider processes that are not output-divergent, every trace of a process is a prefix of a quiescent trace[5].

Work on using a centralised tester also assumes that observations are made in quiescent states [Petrenko et al. 2003; Simão and Petrenko 2011] and we now explain how an implementation relation (that we call $\textbf{dioco}_c$) can be defined; previous work has not formally defined such an implementation relation. Work on asynchronous testing used the notion of a **delay** operator that can be defined in the following way [Huo and Petrenko 2004].

Definition 3.3. Given trace $\sigma \in \mathcal{A}ct^*$, $\textbf{delay}(\sigma)$ is the smallest set of traces defined by the following rules.

(1) $\sigma \in \textbf{delay}(\sigma)$
(2) If $\sigma_1 a^1 a^2 \sigma_2 \in \textbf{delay}(\sigma)$, $a^1 \in O$, $a^2 \in I$ then $\sigma_1 a^2 a^1 \sigma_2 \in \textbf{delay}(\sigma)$

Given set A of traces, we let $\textbf{delay}(A) = \cup_{\sigma \in A}\textbf{delay}(\sigma)$.

The idea is that if a trace σ is one that can be produced by an IOTS that we are communicating with through asynchronous FIFO channels then the output can be observed later than it was produced by the IOTS; it can be delayed sufficiently to be observed after later input is sent. This is illustrated in Figure 3 in which time progresses as we go down the lines that represent the processes (the tester and the SUT), with this showing that $?i?i'!o \in \textbf{delay}(?i!o?i')$. Thus, given a trace σ we have that $\textbf{delay}(\sigma)$ is the set of traces that might be observed if the SUT produces σ. For example, $!r_U!r_L?u_0$ is a trace of M_0 and we have that $\textbf{delay}(!r_U!r_L?u_0) = \{!r_U!r_L?u_0, !r_U?u_0!r_L, ?u_0!r_U!r_L\}$.

In distributed testing with a centralised tester and asynchronous channels there are separate channels to the different ports. Thus, without additional timing information,

[5]Recent work has generalised **dioco** to the case where processes can be output-divergent and has done this by considering infinite traces [Hierons et al. 2012].

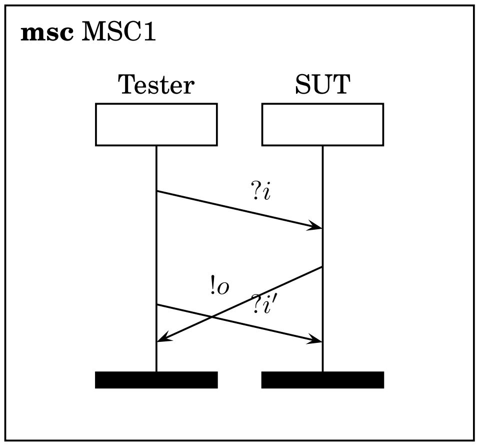

Fig. 3. A delay in testing a single-port system

we have relatively little information regarding the relative order of events at *different* ports: all we know is that if input $?i$ is sent after an output $!o$ is observed then $?i$ is received by the SUT after $!o$ was produced. This leads to a generalised **delay** operator parameterised by the port set.

Definition 3.4. Given port set $\mathcal{P}$ and $\sigma \in \mathcal{A}ct^*$, $\mathbf{delay}_{\mathcal{P}}(\sigma)$ is the smallest set of traces defined by the following rules.

(1) $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma)$
(2) If $\sigma_1 a^1 a^2 \sigma_2 \in \mathbf{delay}_{\mathcal{P}}(\sigma)$ then we have that $\sigma_1 a^2 a^1 \sigma_2 \in \mathbf{delay}_{\mathcal{P}}(\sigma)$ if one of the following holds:
   (a) We have that a^1 and a^2 are at the same port, $a^1 \in O$, and $a^2 \in I$
   (b) We have that a^1 and a^2 are at different ports and either $a^1 \in O$ or $a^2 \in I$.

Given set A of traces, we let $\mathbf{delay}_{\mathcal{P}}(A) = \cup_{\sigma \in A} \mathbf{delay}_{\mathcal{P}}(\sigma)$.

If global trace σ is produced by the SUT then $\mathbf{delay}_{\mathcal{P}}(\sigma)$ is the set of observations that might be made. For example, $!r_U!r_L?l_0?u_0$ is a trace of M_0 and we have that $?l_0!r_L!r_U?u_0 \in \mathbf{delay}_{\{U,L\}}(!r_U!r_L?l_0?u_0)$. This is because $!r_U?l_0!r_L?u_0 \in \mathbf{delay}_{\{U,L\}}(!r_U!r_L?l_0?u_0)$ (rule 2a), $?l_0!r_U!r_L?u_0 \in \mathbf{delay}_{\{U,L\}}(!r_U?l_0!r_L?u_0)$ (rule 2b), and $?l_0!r_L!r_U?u_0 \in \mathbf{delay}_{\{U,L\}}(?l_0!r_U!r_L?u_0)$ (rule 2b).

The above definition adds the ability for an event a^1 to be delayed so that it is observed later than event a^2, which is at a different port, as long as we do not have that $a^1 \in I$ and $a^2 \in O$. The rules thus capture the following situations in which we can swap a^1a^2 when a^1 and a^2 are at different ports.

(1) a^1 and a^2 are inputs: since there are separate channels between the different ports of the SUT and the centralised tester, the orders in which they were sent by the tester and received by the SUT may be different. This case it shown in Figure 4.
(2) a^1 and a^2 are outputs: since there are separate channels between the different ports of the SUT and the centralised tester, the orders in which they were received by the tester and sent by the SUT may be different. This case it shown in Figure 5.
(3) a^1 is an output and a^2 is an input: if output a^1 is produced by the SUT before input a^2 is received, it is possible that the tester observes the output after it sends the input. This case it shown in Figure 6.

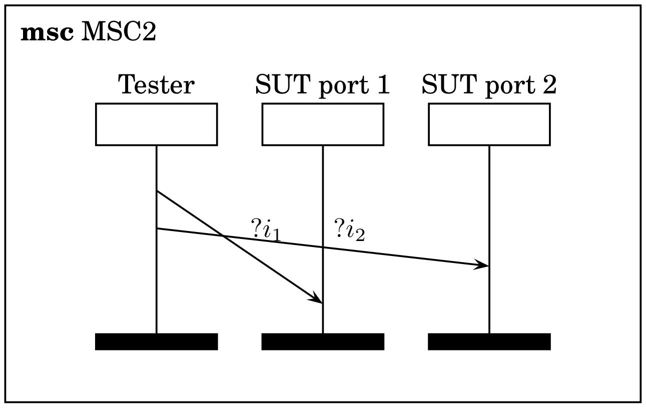

Fig. 4. Order of inputs swapped

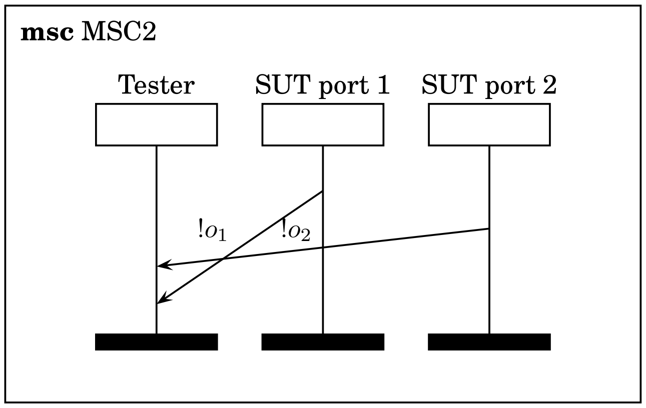

Fig. 5. Order of outputs swapped

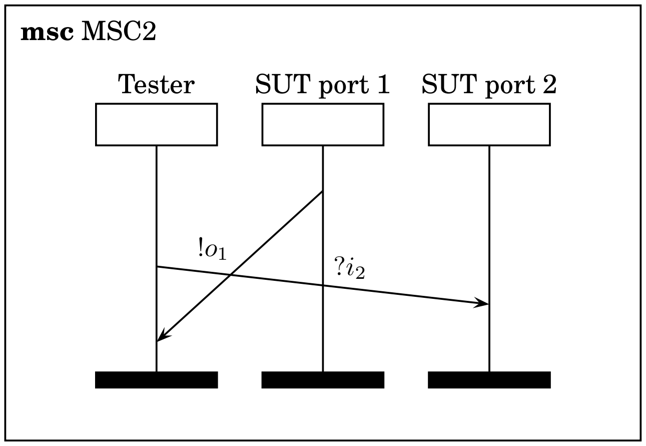

Fig. 6. Order of outputs swapped

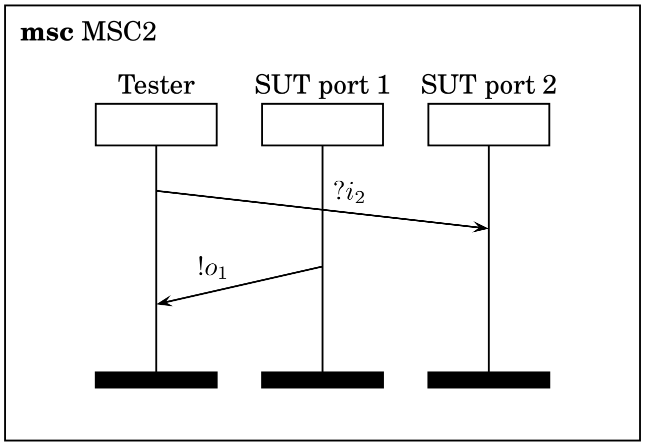

Fig. 7. Input followed by output

The rules do not allow a^1 and a^2 to be swapped in the following cases.

(1) Both values are inputs at a common port p. Here the order is preserved as a result of communications being FIFO.
(2) Both values are outputs at a common port p. Here the order is preserved as a result of communications being FIFO.
(3) The values a^1 and a^2 are at different ports, a^1 is an input, and a^2 is an output. In this case the SUT receives input a^1 and then produces output a^2. In addition, the input is sent by the tester before it is received by the SUT, and the output is sent by the SUT before it is received by the tester. Thus, the tester sends a^1 before it receives a^2. This case it shown in Figure 7.

We can now define an implementation relation for centralised testing through asynchronous channels. This is not quite the same as the implementation relation implicit in the previous work [Simão and Petrenko 2011] since this previous work assumes that inputs are received by the SUT in the order they are sent even if they are sent to different ports. The paper notes that this assumption can be enforced in testing if we can determine when the input queues are empty; we only send inputs when all input queues are empty [Simão and Petrenko 2011]. However, it seems likely that many networks used by real systems will not allow the tester to determine when the network channels are empty. As a result, we do not assume that we can determine when the input queues are empty even though this will be possible for some systems. Note that an additional consequence of the earlier approach [Simão and Petrenko 2011] was that it was only possible to apply inputs in quiescent states and this clearly restricts testing since it does not allow some behaviours of the SUT to be explored.

First we define an implementation relation for the case in which it is possible to observe quiescence.

Definition 3.5. Given IOTSs N and M with the same input and output alphabets we have that N $\mathbf{dioco}_c$ M if for every trace $\sigma \in \mathcal{A}ct^*$ with $\sigma\delta \in \mathcal{T}r(N)$ we have that there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$.

This requires that for every quiescent trace σ of the SUT there is a quiescent trace σ' of M such that σ is an observation that might be made if the SUT produces σ'; this is because $\mathbf{delay}_{\mathcal{P}}(\sigma')$ is the set of observations that might be made if the SUT produces

σ'. For example, if we produce an IOTS M_0' from M_0 by swapping the labels ($!r_U$ and $!r_L$) of the transitions on the path of length 2 from s_0 then we have that M_0' $\mathbf{dioco}_c$ M_0.

Note that Definition 3.5 only refers to the traces of N and not to the set of observations that a centralised tester might make when interacting with N. However, the actual observation need not be a trace of the SUT: if the SUT produces trace σ then the observation could be any element of $\mathbf{delay}_{\mathcal{P}}(\sigma)$. Thus, we might instead have required that for every trace σ'' in $\mathbf{delay}_{\mathcal{P}}(\sigma)$ there is some $\sigma' \in \mathcal{T}r(M)$ such that σ'' is in $\mathbf{delay}_{\mathcal{P}}(\sigma')$. The following shows that all of these observations are allowed if N $\mathbf{dioco}_c$ M and so this would lead to a definition that is equivalent to Definition 3.5.

PROPOSITION 3.6. *Let us suppose that N and M are IOTSs with the same input and output alphabets such that N $\mathbf{dioco}_c$ M and also that $\sigma \in \mathcal{A}ct^*$ is a quiescent trace of N. If $\sigma'' \in \mathbf{delay}_{\mathcal{P}}(\sigma)$ then there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma'' \in \mathbf{delay}_{\mathcal{P}}(\sigma')$.*

PROOF. Since N $\mathbf{dioco}_c$ M, there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$. But, by the definition of $\mathbf{delay}_{\mathcal{P}}$, since $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$ we have that all elements of $\mathbf{delay}_{\mathcal{P}}(\sigma)$ are in $\mathbf{delay}_{\mathcal{P}}(\sigma')$. The result thus follows. □

It is straightforward to adapt $\mathbf{dioco}_c$ to the case where we do not observe quiescence.

Definition 3.7. Given IOTSs N and M with the same input and output alphabets we have that N $\mathbf{dioco}_c'$ M if for every trace $\sigma \in (\mathcal{A}ct \setminus \{\delta\})^*$ with $\sigma\delta \in \mathcal{T}r(N)$ we have that there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$.

4. A HYBRID FRAMEWORK

Now that we have defined implementation relations for the centralised and distributed approaches it is possible to formally compare them. First we show that an SUT N might conform to the specification M in centralised testing but not in distributed testing. This shows that there are behaviours of possible SUTs that are not behaviours of the specification, and so can be seen as failures, where this can be identified in distributed testing but not in centralised testing.

PROPOSITION 4.1. *There are IOTSs N and M with the same input and output alphabets such that N $\mathbf{dioco}_c$ M but where we do not have that N $\mathbf{dioco}$ M.*

PROOF. Consider the IOTSs M_1 and N_1 shown in Figure 8 in which there is only one port. The quiescent traces of N_1 are all those in the sets $\{\epsilon\}$, $\{?i!o\}\{?i\}^*$ and $\{?i?i!o\}\{?i\}^*$. To see that N_1 $\mathbf{dioco}_c$ M_1 it is sufficient to observe that ϵ and $?i!o$ are quiescent traces of M_1 and that $?i?i!o$ can be formed from the quiescent trace $?i!o?i$ of M_1 by delaying output. However, the quiescent trace $?i?i!o$ of N_1 is not equivalent to a trace of M_1 under $\sim$ and so we have that N_1 does not conform to M_1 under $\mathbf{dioco}$ as required. □

The following shows that there are also cases where the centralised approach is more effective than distributed testing. This uses Figure 9 in which an arc with multiple labels corresponds to multiple transitions (one per label). For example, the arc from the initial state of M_2 with labels $?i_1, ?i_2$ corresponds to two transitions: one with label $?i_1$ and one with label $?i_2$.

PROPOSITION 4.2. *There are IOTSs N and M with the same input and output alphabets such that N $\mathbf{dioco}$ M but where we do not have that N $\mathbf{dioco}_c$ M.*

PROOF. Consider the IOTSs M_2 and N_2 shown in Figure 9 in which there are two ports. A quiescent trace of N_2 is either ϵ or a sequence formed from the following by adding a (possibly empty) suffix that is an input sequence: $?i_2$, $?i_1?i_1$, $?i_1?i_2$,

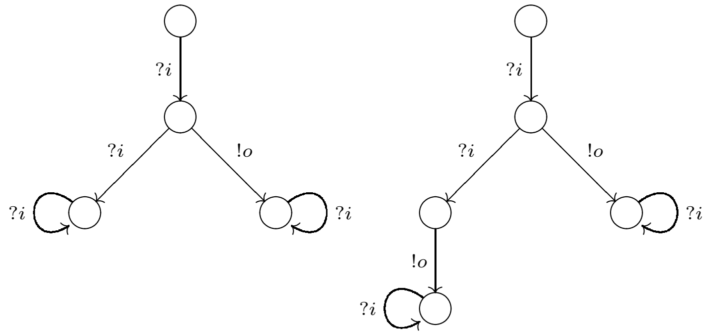

Fig. 8. IOTSs M_1 (left) and N_1 (right)

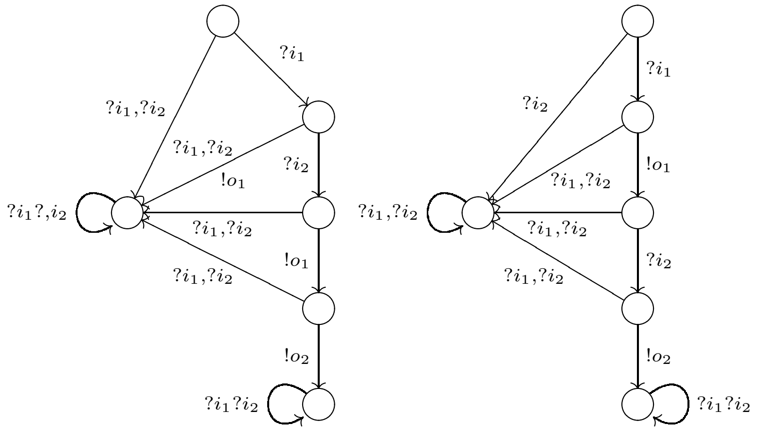

Fig. 9. IOTSs M_2 (left) and N_2 (right)

$?i_1!o_1$, $?i_1!o_1?i_2!o_2$. However, ϵ, $?i_1?i_1$, $?i_1?i_2$ and $?i_1!o_1$ are quiescent traces of M_2 and $?i_1!o_1?i_2!o_2$ is equivalent to the quiescent trace $?i_1?i_2!o_1!o_2$ of M_2 under $\sim$. Thus, we have that N_2 **dioco** M_2.

To see that N_2 $\mathbf{dioco}_c$ M_2 does not hold it is sufficient to consider the quiescent trace $?i_1!o_1?i_2!o_2$ of N_2. The only quiescent trace of M_2 that contains these inputs and outputs is $?i_1?i_2!o_1!o_2$ and $?i_1!o_1?i_2!o_2 \notin \mathbf{delay}_{\mathcal{P}}(?i_1?i_2!o_1!o_2)$. The result therefore holds. □

We therefore have that **dioco** and $\mathbf{dioco}_c$ are incomparable and can distinguish different potential SUTs from the specification. The same examples show that $\mathbf{dioco}'$ and $\mathbf{dioco}'_c$ are also incomparable. This suggests that we should consider a hybrid approach. Under this hybrid approach there is a centralised tester that interacts asynchronously with the SUT and at each port p there is a local tester. The centralised tester is responsible for sending input to the SUT and receives outputs. However, the local tester at port p logs the sequence of observations made at p. Under this hybrid approach the centralised tester observes a trace as with $\mathbf{dioco}_c$ but, in addition, the local projections of the global trace produced by the SUT are recorded.

The hybrid framework is outlined in Figure 10 in which N is the SUT, the T_p are local testers, and T is a centralised tester. Dotted lines denote asynchronous communications and solid lines denote synchronous communications. The observation made in

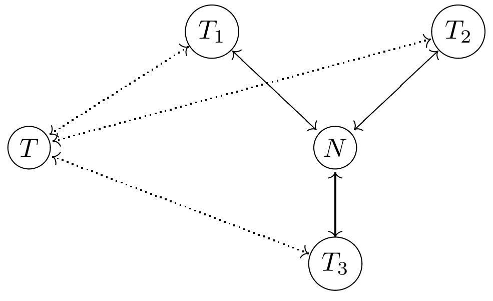

Fig. 10. Hybrid Test Architecture

testing consists of one trace that contains all of the events, observed by T, in addition to the local traces. Thus, if the SUT produces global trace σ then the tester at port $p \in \mathcal{P}$ observes $\pi_p(\sigma)$ and the centralised tester T observes some element of $\mathbf{delay}_{\mathcal{P}}(\sigma)$. For example, if the SUT produced trace $?i_1!o_2?i_2$ then the tester at port 1 observes $?i_1$, the tester at port 2 observes $!o_2?i_2$, and the centralised tester might observe $?i_1?i_2!o_2$.

The hybrid and centralised approaches are identical when it comes to applying test cases: they only differ in the observations made. Let us suppose again that we test from M_0 (Figure 2) and we are interested in the trace $\sigma = !r_U!r_L?l_1?u_1!1_U!1_L$ and the SUT produces $\sigma'' = !r_U!r_L!1_U!1_L?l_1?u_1$. As with the centralised approach, the central tester might still observe σ. However, the local testers will observe local traces $!r_U!1_U?u_1$ and $!r_L!1_L?l_1$; once we gather together the test logs we will be able to determine that the trace was not σ.

The effectiveness of the hybrid framework is represented by the following implementation relation.

Definition 4.3. Given IOTSs N and M with the same input and output alphabets we have that N $\mathbf{dioco}_s$ M if for every trace $\sigma \in \mathcal{A}ct^*$ with $\sigma\delta \in \mathcal{T}r(N)$ we have that there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma' \sim \sigma$ and $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$.

This essentially requires that the observation made, which is a set of local traces plus a global trace observed by the centralised tester, is also one that can be made when interacting with the specification.

If we cannot observe quiescence during testing then we obtain the following variation.

Definition 4.4. Given IOTSs N and M with the same input and output alphabets we have that N $\mathbf{dioco}'_s$ M if for every trace $\sigma \in (\mathcal{A}ct \setminus \{\delta\})^*$ with $\sigma\delta \in \mathcal{T}r(N)$ we have that there exists a quiescent trace $\sigma' \in \mathcal{T}r(M)$ such that $\sigma' \sim \sigma$ and $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$.

The following shows how $\mathbf{dioco}_s$ relates to $\mathbf{dioco}$ and $\mathbf{dioco}_c$.

PROPOSITION 4.5. *Given IOTSs N and M with the same input and output alphabets, if we have that N $\mathbf{dioco}_s$ M then we also have that both N $\mathbf{dioco}$ M and N $\mathbf{dioco}_c$ M hold. In addition, it is possible for N $\mathbf{dioco}$ M to hold and for N $\mathbf{dioco}_s$ M not to hold or for N $\mathbf{dioco}_c$ M to hold and for N $\mathbf{dioco}_s$ M not to hold.*

PROOF. The first part is immediate from the definition. The second part follows in the same way as Propositions 4.1 and 4.2. □

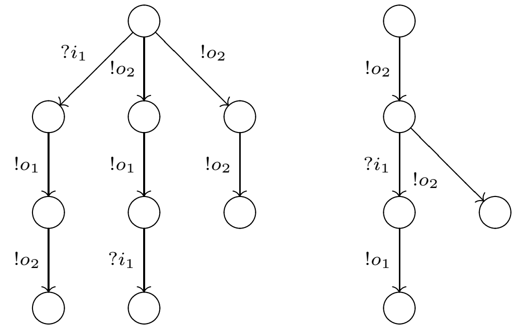

Fig. 11. IOTSs M_3(left) and N_3 (right)

Interestingly, $\mathbf{dioco}_s$ need not hold even if **dioco** and $\mathbf{dioco}_c$ both hold. This tells us that the hybrid framework is more effective that separately applying both the centralised and distributed approaches.

PROPOSITION 4.6. *Given IOTSs N and M with the same input and output alphabets, it is possible for both N* **dioco** *M and N* $\mathbf{dioco}_c$ *M to hold and for N* $\mathbf{dioco}_s$ *M not to hold.*

PROOF. Consider the IOTSs M_3 and N_3 shown in Figure 11 in which self-loops labelled with input are implicit (in order to make the processes input-enabled). The quiescent traces of N_3 are $!o_2!o_2$ and traces that can be formed by adding to $!o_2?i_1!o_1$ a suffix in the form of a (possibly empty) input sequence. Clearly $!o_2!o_2$ is a quiescent trace of M_3 and so it is sufficient to consider traces starting with $!o_2?i_1!o_1$.

Now consider the three implementation relations. We have that N_3 **dioco** M_3 since $!o_2?i_1!o_1 \sim ?i_1!o_1!o_2$ and $?i_1!o_1!o_2$ is a trace of M_3. We have that N_3 $\mathbf{dioco}_c$ M_3 since M_3 has the trace $!o_2!o_1?i_1$ and $!o_2?i_1!o_1 \in \mathbf{delay}_{\mathcal{P}}(!o_2!o_1?i_1)$. However, the trace $!o_2?i_1!o_1$ is not allowed under $\mathbf{dioco}_s$ since we require a trace σ that is equivalent to $!o_2?i_1!o_1$ under $\sim$, the only such trace is $?i_1!o_1!o_2$ but $!o_2?i_1!o_1 \notin \mathbf{delay}_{\mathcal{P}}(?i_1!o_1!o_2)$. The result therefore holds. □

It is straightforward to see that corresponding results also hold for $\mathbf{dioco}'$, $\mathbf{dioco}'_c$ and $\mathbf{dioco}'_s$.

5. THE ORACLE PROBLEM

The Oracle problem is that of deciding whether an observation is allowed by the specification. It is known that the Oracle problem is NP-hard when using the **dioco** implementation relation [Hierons 2012b]. In this section we prove that it is also NP-hard to determine whether an observation is one allowed by a specification when using $\mathbf{dioco}_c$ or $\mathbf{dioco}_s$. We prove that the Oracle problem is NP-hard by showing that we can reduce the following problem to it.

Definition 5.1. Given boolean variables $z_1, \dots, z_r$ let $C_1, \dots, C_k$ 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 considers the case where a proposition is written in conjunctive normal form $C_1 \wedge \dots \wedge C_k$, each C_i being the disjunction of three literals. The one-in-three SAT problem is known to be NP-complete [Schaefer 1978].

If we do not place restrictions on M then the Oracle problem is undecidable even for synchronous testing. We therefore restrict attention to the case where M has finite sets of states, inputs and outputs. The following is adapted from the proof of the Oracle problem being NP-hard for **dioco** [Hierons 2012b].

THEOREM 5.2. *Let us suppose that M is an IOTS with finite sets of states, inputs and outputs. Given trace $\sigma \in \mathcal{A}ct^*$, the problem of deciding whether there exists a quiescent trace σ' of M such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$ is NP-complete.*

PROOF. First we prove that the problem is in NP. In polynomial time we can randomly choose some σ' such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$. It is then possible to determine whether σ' is a quiescent trace of M in polynomial time: this corresponds to solving the membership problem for a finite automaton. Thus, a nondeterministic Turing machine can solve this problem in polynomial time by guessing a σ' and then checking that it is allowed by M and so the problem is in NP.

We now prove that this problem is NP-hard. The proof will operate by showing that the one-in-three SAT problem can be reduced to this problem. We thus suppose that we have boolean variables $z_1, \ldots, z_r$ and clauses $C_1, \ldots, C_k$. We will define an IOTS M 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 $r+1, \ldots, r+k$.

IOTS M has initial state s_0 and input $?i_p$ $(1 \leq p \leq r)$ in this state can lead to either of the following.

(1) A transition with input $?i_p$ that is followed by a sequence of transitions, returning to s_0, that (between them) send output $!o_{r+j}$ to port $r+j$ if and only if C_j contains literal z_p and otherwise send no output to port $r+j$ $(1 \leq j \leq k)$.
(2) A transition with input $?i_p$ that is followed by a sequence of transitions, returning to s_0, that (between them) send output $!o_{r+j}$ to port $r+j$ if and only if C_j contains literal $\neg z_p$ and otherwise send no output to port $r+j$ $(1 \leq j \leq k)$.

If input is received in any of the states in these cycles, other than s_0, then M moves to error state s_e from which it produces output $!o_e$ at port 1 and then moves to a state s_e' that has only self-loop transitions labelled with inputs. There are also self-loop transitions labelled with inputs in state s_e in order to make M input-enabled.

Now consider the trace $\sigma =?i_1?i_2 \ldots ?i_r!o_{r+1} \ldots !o_{r+k}$. Since output $!o_e$ is not produced, if there exists a quiescent trace σ' of M such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$ then each input $?i_p$ in σ' is received once and in state s_0. Since σ' is a quiescent trace, σ' must be able to take M from s_0 to s_0. As a result, on receiving $?i_p$, either an output is sent to all ports that correspond to clauses that contain literal z_p or output is sent to all ports that correspond to clauses that contain literal $\neg z_p$. Thus, there exists quiescent $\sigma' \in \mathcal{T}r(M)$ such that $\sigma \in \mathbf{delay}_{\mathcal{P}}(\sigma')$ if and only if there exists an assignment to the boolean variables $z_1, \ldots, z_r$ such that each C_i contains exactly one true literal. The result thus follows from the one-in-three SAT problem being NP-hard and it being possible to construct M and σ in polynomial time. □

THEOREM 5.3. *The Oracle problem is NP-complete for $\mathbf{dioco}_c$, $\mathbf{dioco}_c'$, $\mathbf{dioco}_s$, and $\mathbf{dioco}_s'$.*

PROOF. The result for $\mathbf{dioco}_c$ follows immediately from Theorem 5.2. For $\mathbf{dioco}_s$, it is sufficient to observe that the example used in the proof of Theorem 5.2 only has one observation at each port and so the notions of observation are equivalent for the centralised and hybrid approaches. Finally, the proofs for $\mathbf{dioco}_c'$ and $\mathbf{dioco}_s'$ follow in the same way since we do not observe quiescence before the end of the trace. □

We now consider the case where there is an upper bound b on the number of ports and show that this allows us to define a polynomial time algorithm that solves the Oracle problem. This result suggests that the Oracle problem being NP-complete should not be a practical barrier for classes of systems such as communications protocols where there are only a few ports (typically two in the case of communications protocols).

Let us suppose that the centralised tester observes trace σ and the local tester at port p observes σ_p ($p \in \mathcal{P}$). Let us also suppose that all inputs and outputs are labelled by the tester in order to make them unique; this will allow us to place a partial order over a set of elements, rather than a multi-set. This can be achieved by, for example, replacing the occurrence of an observation a with the pair (a, k) such that there have been exactly $k-1$ previous observations of a. Thus, for example, if the centralised tester observes $?i_1!o_1?i_2?i_1$ then we would use the trace $(?i_1, 1)(!o_1, 1)(?i_2, 1)(?i_1, 2)$. Below we show how we can define a finite automaton $M(\sigma, \sigma_1, \ldots, \sigma_n)$ that accepts all traces that the SUT might have produced given the observations made. Once we have done this, we simply ask whether there is any common trace in the languages defined by M and $M(\sigma, \sigma_1, \ldots, \sigma_n)$, a problem that can be solved in time that is polynomial in the number of states of M and $M(\sigma, \sigma_1, \ldots, \sigma_n)$.

Let $E(\sigma)$ denote the set of observations (events) in σ, labelled to make them unique if necessary. Given σ and the σ_p ($p \in \mathcal{P}$) we define a partial order $\ll$ on the observations in $E(\sigma)$ such that given $e, e' \in E(\sigma)$ we know that the SUT *must* have produced e before e' if and only if $e \ll e'$. The partial order $\ll$ is defined as follows.

Definition 5.4. Let us suppose that the central tester observes $\sigma = e_1 \ldots e_k$ and the local trace observed at port p is σ_p ($p \in \mathcal{P}$). For port p let $<_p$ denote the ordering of the events at p in σ_p. The partial order $\ll$ is the transitive closure of the following, in which $e_i, e_j \in E(\sigma)$.

(1) $e_i \ll e_j$ if there exists port $p \in \mathcal{P}$ such that $e_i, e_j \in \mathcal{A}ct_p$ and $e_i <_p e_j$;
(2) $e_i \ll e_j$ if e_i and e_j are at different ports, $i < j$, $e_i \in O$, and $e_j \in I$.

The following is immediate from the definition and says that the events ordered under $\ll$ must have been produced in the same order by the SUT.

PROPOSITION 5.5. *If in testing the centralised tester observes trace σ and the local tester at port p observes σ_p (all $p \in \mathcal{P}$) then the SUT must have produced a trace $e_1 \ldots e_k$ that is a permutation of σ such that if $e_i \ll e_j$ then $i < j$.*

From this it is clear also that if σ is a trace that the SUT might have produced then for every prefix σ' of σ we have that the set of observations $E(\sigma')$ in σ' is downwardly closed under $\ll$: if $e \in E(\sigma')$ and $e' \ll e$ then $e' \in E(\sigma')$. This is the key property: we will define a finite automaton whose states correspond to the downwardly closed subsets of $E(\sigma)$. First we briefly say what a finite automaton is.

Definition 5.6. A finite automaton N is defined by a tuple (Q, A, U, q_o, F) such that Q is the finite set of states, A is the finite alphabet, $U \subseteq Q \times A \times Q$ is the set of transitions, $q_0 \in Q$ is the initial state, and F is the set of final states.

A finite automaton $N = (Q, A, U, q_o, F)$ defines the language $L(N)$ of labels of paths that start at the initial state of N and end at a final state from F.

Definition 5.7. The finite automaton $M(\sigma, \sigma_1, \ldots, \sigma_n)$ is defined by the tuple $(Q, I \cup O, U, q_o, F)$ in which Q is the set of subsets of $E(\sigma)$ that are downwardly closed under $\ll$, q_0 is the empty set, F contains one final state which is $E(\sigma)$, and $(q, a, q') \in U$ if and only if $q' = q \cup \{a\}$.

The following is immediate from Proposition 5.5.

PROPOSITION 5.8. *$L(M(\sigma, \sigma_1, \ldots, \sigma_n))$ is the set of traces that the SUT might have produced if in testing the centralised tester observes trace σ and the local tester at port p observes σ_p (all $p \in \mathcal{P}$).*

We therefore have the following result.

THEOREM 5.9. *The Oracle problem when we have a centralised tester, local testers, an SUT with at most b ports, and an IOTS specification with at most ℓ transitions can be solved in time of $O(\ell b|\sigma|^b)$.*

PROOF. By Proposition 5.8, given σ and the σ_p ($p \in \mathcal{P}$), it is sufficient to determine whether there is a trace of M that is accepted by $M(\sigma, \sigma_1, \ldots, \sigma_n)$. Given two finite automata with k_1 and k_2 transitions one can decide whether their languages contain a common word by taking their product and performing a depth-first search for a final state. Since the product machine has $O(k_1 k_2)$ transitions and a depth-first search can be performed in linear time [Tarjan 1972], this can be achieved in $O(k_1 k_2)$ time.

Now consider the finite automaton $M(\sigma, \sigma_1, \ldots, \sigma_n)$. A state of $M(\sigma, \sigma_1, \ldots, \sigma_n)$ corresponds to a downwardly closed subset of $E(\sigma)$. Observe that for a port p, all events at p in $E(\sigma)$ are ordered under $\ll$ (forming a chain). Thus, every downwardly closed subset of $E(\sigma)$ is defined by a set E' of events and all events below these under $\ll$ and where for each port p we have that E' contains at most one event. Thus, $M(\sigma, \sigma_1, \ldots, \sigma_n)$ has $O(|\sigma|^b)$ states. Further, for a state q and port p, since all events at p in $E(\sigma)$ are ordered under $\ll$ there is only one possible next event at p. Thus, for each state q there are at most b transitions leaving q. Thus, $M(\sigma, \sigma_1, \ldots, \sigma_n)$ has $O(b|\sigma|^b)$ transitions. The result therefore follows. $\square$

While this complexity is exponential in terms of b, if we have an upper bound on the number of ports then the complexity is polynomial. This result is relevant for application domains where there are usually only a few ports. For example, for communications protocols there are typically two ports representing the two agents that are exchanging information. The distributed test architecture [ISO/IEC 1995], as standardised by ISO, has two ports U and L for this situation. Ports U and L represent the upper interface and lower interface respectively. The tester at U is an agent that interacts with the implementation of the protocol on a machine A (and so acts as a layer above the SUT in the protocol stack), while the tester at L is on a separate machine and interacts with the SUT through a physical network. See, for example, [Sarikaya and von Bochmann 1984; Dssouli and von Bochmann 1985; 1986].

6. REACHING AND DISTINGUISHING STATES

It is known that it is generally undecidable whether there is a distributed test case consisting of independent testers that is guaranteed to force a deterministic finite state machine M into a given state s or to distinguish two states s, s' of M [Hierons 2010]. Naturally, these results immediately extend to IOTSs even if we restrict the sets of states, inputs, and outputs to being finite. In this section we consider the corresponding problems for testing against an IOTS M using a single centralised tester that interacts asynchronously with M through FIFO channels. The IOTS M used in the proof will not produce any output before the objective is achieved and so we will see that the results apply to both the centralised and hybrid approaches. We start by considering the problem of reaching a state; the result regarding distinguishing states then follows easily.

The proof, that reachability is undecidable for distributed testing (as opposed to centralised testing or testing in the hybrid framework) [Hierons 2010], used the result that it is undecidable whether there is a winning strategy in multi-player games [Peterson and Reif 1979; Demaine and Hearn 2008]. In this section we adapt the proof of

Demaine and Hearn [Demaine and Hearn 2008], that it is undecidable whether there is a winning strategy in a multi-player game, to the problem of deciding whether there is a test case that is guaranteed to move an IOTS M into state s when using a centralised tester that interacts with M through asynchronous channels. The proof also has some conceptual similarities to Floyd's proof that Post's Correspondence Problem is undecidable [Floyd 1964], as described by Davis and Weyuker [Davis and Weyuker 1993].

We will assume that we have been given a Turing Machine $\mathcal{TM}$ that halts on an empty tape (it is said to halt if it reaches halting state h and the tape is empty). We will show how a 2-port IOTS $M_{\mathcal{TM}}$ can be constructed such that deciding whether there is a test case guaranteed to take $M_{\mathcal{TM}}$ to a particular state h_M is equivalent to deciding whether $\mathcal{TM}$ halts, a problem that is known to be undecidable. All paths of $M_{\mathcal{TM}}$ will contain only inputs as labels and so in testing we gain nothing through being adaptive: the tester has no observations to use as the basis of adapting its behaviour. In addition, the hybrid and centralised approaches are equivalent. A test case will thus correspond to an input sequence σ and if $\sigma \in I^*$ is used then $M_{\mathcal{TM}}$ will receive an input sequence that is equivalent to σ under $\sim$. An important property is that we cannot know which $\sigma' \sim \sigma$ is received by $M_{\mathcal{TM}}$ and so for σ to be guaranteed to reach h_M we require that all $\sigma' \sim \sigma$ are guaranteed to take $M_{\mathcal{TM}}$ to h_M. We will define $M_{\mathcal{TM}}$ with a state h_M such that given an input sequence σ, the input sequence received by $M_{\mathcal{TM}}$ is guaranteed to reach h_M if and only if the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ define the same halting computation of $\mathcal{TM}$.

A computation of $\mathcal{TM}$ is a sequence of consecutive configurations, starting with the initial configuration. A configuration of a Turing machine is defined by the current state, the tape contents, and the location of the tape head. We use the position of the state name, in a sequence defining a configuration, to indicate the location of the tape head. A sequence of the form $\sigma q \sigma'$ thus represents a configuration in which the state is q, the tape contains $\sigma\sigma'$ and the tape head is on the cell represented by the first element of σ'. The input set of $M_{\mathcal{TM}}$ will contain the special values $\#_1 \in I_1$ and $\#_2 \in I_2$ that will be used to separate configurations and also γ_1 and γ_2 that will be used to mark the end of a computation. Given Turing machine $\mathcal{TM}$ with state set Q and alphabet Σ, $M_{\mathcal{TM}}$ will have (disjoint) input alphabets $I_1 = \Sigma_1 \cup Q_1 \cup \{\#_1, \gamma_1\}$ and $I_2 = \Sigma_2 \cup Q_2 \cup \{\#_2, \gamma_2\}$ in which Σ_1 and Σ_2 are sets of copies of the elements of Σ and Q_1 and Q_2 are sets of copies of the elements of Q. We use labels to ensure that the sets $\Sigma_1, \Sigma_2, Q_1, Q_2, \{\#_1, \gamma_1\}, \{\#_2, \gamma_2\}$ are pairwise disjoint.

Now consider two consecutive configurations c_1 and c_2 in a computation of $\mathcal{TM}$ and assume that c_1 is represented by $\sigma_1 q_1 \sigma_1'$ and c_2 is represented by $\sigma_2 q_2 \sigma_2'$. The allowed differences between the sequences representing these configurations are defined by the transitions of $\mathcal{TM}$ and cannot be more than three letters since at most the contents of the cell that the tape head is on is changed and the tape head moves one place. As a result, if we are given two sequences representing configurations c_1 and c_2 interleaved so that the values of c_1 and c_2 alternate (e.g. it starts with the first value from c_1, followed by the first value from c_2, then the second value from c_1 etc.) then we can check in finite (and bounded) space whether there is a transition that takes $\mathcal{TM}$ from configuration c_1 to configuration c_2. This process involves looping while the values in the sequences are the same, then checking that the sequences of three values from the first change are consistent with the transitions of $\mathcal{TM}$, and finally checking that there are no additional changes. Thus, if we are given the representations of c_1 and c_2 with values alternating then we can define an IOTS that checks whether c_2 is a configuration that can follow c_1 in a computation of $\mathcal{TM}$. This check will be implemented by the component $CheckConf$ of $M_{\mathcal{TM}}$ described below.

The IOTS $M_{\mathcal{TM}}$ will be constructed from a set of components that operate in parallel, synchronising on common events (inputs and outputs). Each component will have a state set and the target final state h_M is thus a tuple of states: one for each component. We briefly outline the roles of these components before defining them.

(1) $Start_p$, $1 \leq p \leq 2$, is a component that ends in the desired state s_p that forms part of h_M if and only if the projection on p of the input sequence received starts with the initial configuration of $\mathcal{TM}$. Thus, an input sequence σ is guaranteed to take $M_{\mathcal{TM}}$ to a state where $Start_p$ is in state s_p, $1 \leq p \leq 2$, if and only if the $\pi_1(\sigma)$ and $\pi_2(\sigma)$ both start with representations of the initial configuration of $\mathcal{TM}$.
(2) End_p, $1 \leq p \leq 2$, is a component that ends in the desired state e_p that forms part of h_M if and only if the projection on p of the input sequence received ends with a representation of the halting configuration of $\mathcal{TM}$ followed by γ_p. Thus, an input sequence σ is guaranteed to take $M_{\mathcal{TM}}$ to a state where End_p is in state e_p, $1 \leq p \leq 2$, if and only if $\pi_1(\sigma)$ and $\pi_2(\sigma)$ both end with representations of the halting configuration of $\mathcal{TM}$ followed by γ_1 and γ_2 respectively.
(3) $CheckSeq$ is a component that is guaranteed to end in state h_S when given input sequence σ if and only if $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent (they are the same if labels are removed) and end with γ_1 and γ_2 respectively.
(4) The component $CheckConf$ relies on the projections $\pi_1(\sigma)$ and $\pi_2(\sigma)$ being equivalent and to start with representations of the initial configuration (checked by $Start_1$, $Start_2$, and $CheckSeq$). In this case the component $CheckConf$ is guaranteed to end in state h_C if and only if $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent the same computation of $\mathcal{TM}$: a sequence of configurations $c_1 \ldots c_k$ such that for all $1 \leq i < k$ we have that $\mathcal{TM}$ can move from c_i to c_{i+1}.

The basic idea is that if input sequence σ is guaranteed to take $M_{\mathcal{TM}}$ to h_M then we have that the following hold.

(1) Both $\pi_1(\sigma)$ and $\pi_2(\sigma)$ start with representations of the initial configuration and end with representations of the halting configuration (as a result of $Start_1, Start_2, End_1, End_2$ ending in the desired states).
(2) The projections $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent up to labelling, as a result of $CheckSeq$ ending in the desired state.
(3) The projections $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent the same computation of $\mathcal{TM}$, as a result of the above and $CheckConf$ ending in the desired state.

Thus, if an input sequence σ is guaranteed to take $M_{\mathcal{TM}}$ to h_M then we must have that $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent the same halting computation of $\mathcal{TM}$. As a result, there is such an input sequence σ if and only if $\mathcal{TM}$ has a halting computation (an undecidable problem).

We will define $M_{\mathcal{TM}}$ in terms of (IOTS) components acting in parallel. Given components M_1 and M_2, $M_1 \parallel M_2$ will denote M_1 and M_2 acting in parallel and synchronising on observable events. Since $M_{\mathcal{TM}}$ will be formed from the parallel composition of several components, a state of $M_{\mathcal{TM}}$ will be a tuple of the states of these components and whenever an input $?i$ is received by $M_{\mathcal{TM}}$ each component takes a transition with label $?i$. We now describe the components, assuming that Turing machine $\mathcal{TM}$ has been given.

Components $Start_1$ and $Start_2$ check that the projections of the input sequence σ received both start with the unique initial configuration. Once this is confirmed these components move to states s_1 and s_2 and further transitions do not change their state. An input at port q does not change the state of $Start_p$, $q \neq p$. State h_M has these components being in states s_1 and s_2. We have the following property regarding an input sequence σ sent by a centralised tester (and the resultant input sequence σ' received by the SUT).

LEMMA 6.1. *An input sequence $\sigma \in \mathcal{A}ct^*$ sent by a centralised tester is guaranteed to lead to $Start_1 \parallel Start_2$ receiving an input sequence that takes it to state (s_1, s_2) if and only if the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ both start with copies of the initial configuration of $\mathcal{TM}$.*

PROOF. Consider $1 \leq p \leq 2$ and IOTS $Start_p$. By definition, inputs at port $q \neq p$ do not change the state of $Start_p$. By construction σ takes $Start_p$ to state s_p if and only if $\pi_p(\sigma)$ starts with a representation of the initial configuration of $\mathcal{TM}$. The result therefore follows. □

Component End_p, $1 \leq p \leq 2$, checks whether the projection on port p of the trace received ends in the unique halting configuration followed by γ_p; if this is the case then the component End_p is in state e_p. Further input at p after γ_p takes End_p to an error state, as does the input of γ_p if not preceded by a description of the halting configuration. An input at port q does not change the state of End_p, $q \neq p$. State h_M has these components being in states e_1 and e_2.

LEMMA 6.2. *An input sequence $\sigma \in \mathcal{A}ct^*$ sent by a centralised tester is guaranteed to lead to $End_1 \parallel End_2$ receiving an input sequence that takes it to state (e_1, e_2) if and only if the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ both end with copies of the halting configuration of $\mathcal{TM}$ followed by γ_1 and γ_2 respectively.*

PROOF. Consider $1 \leq p \leq 2$ and IOTS End_p. By definition, inputs at port $q \neq p$ do not change the state of End_p. By construction σ takes End_p to state e_p if and only if $\pi_p(\sigma)$ ends with a representation of the halting configuration of $\mathcal{TM}$ followed by γ_p. The result therefore follows. □

There is a component $CheckSeq$ that is responsible for checking that the projections $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent equivalent sequences (they only differ in the labels). This component is designed to be guaranteed to end in state h_S when using $\sigma \in \mathcal{A}ct^*$ if and only if the projections $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent up to labelling. This uses a flag seq, which can take on the values 0, 1, 2, 3, $\perp$, $\perp_1$, and $\perp_2$ and initially is 0. The flag having value 0 represents a situation in which we know that the same number of inputs have been received at the ports. If seq is 0 and input $?i_p$ is received at port $1 \leq p \leq 2$ then $?i_p$ is stored and seq becomes p. If seq is p and input $?i_q$ is received at port $q \neq p$ and $?i_q$ is equivalent to the value $?i_p$ stored then seq becomes 0; if $?i_q$ is not equivalent to $?i_p$ then the component moves to state $\perp$ and cannot then change state (state h_M is then unreachable). Thus, if the sequence $\sigma' \sim \sigma$ received by $M_{\mathcal{TM}}$ does alternate between values at ports 1 and 2 then this component ends in state $seq = 0$ if and only if the two projections are equivalent.

We now explain what happens if seq is 1 and input at port 1 is received or, equivalently, seq is 2 and input at port 2 is received; this can happen even if the two local traces are equivalent. In these cases we no longer compare inputs received at the ports (to ensure that we only need a finite number of states in making comparisons) and seq becomes $\perp_1$ or $\perp_2$ respectively. From $\perp_1$ any input at port 2 moves the component to 3; from $\perp_2$ any input at port 1 moves the component to 3. This avoids the possibility of a test sequence 'cheating', by having more inputs at a port p, by requiring that at least one more input $?i$ is received at the other port. Thus, if the local trace at a port p is longer than the local trace at $q \neq p$ then the component can become stuck in the state $\perp_p$: this is achieved by starting with an input at port q and then alternating between the ports and finally apply the remaining elements from p.

As a result of the above, if the two local traces do not have the same length then there is an interleaving that leads to the component being stuck in either $\perp_1$ or $\perp_2$ and, in addition, the final value can only be $\perp_1$ or $\perp_2$ if the two local traces have different lengths. If the two local traces have the same length then it is possible that the input sequence σ' received has these alternating in which case the final state has $seq = 0$ if and only if the local traces are equivalent.

Thus, in order to ensure that this component ends with seq being either 0 or 3 we require an input sequence with equivalent projections. If γ_1 and γ_2 have been received and seq is either 0 or 3 then the component moves to the special state h_S, which is the state of $CheckSeq$ in h_M. This component therefore has the following property.

LEMMA 6.3. *An input sequence $\sigma \in \mathcal{A}ct^*$ sent by a centralised tester is guaranteed to lead to $CheckSeq$ receiving an input sequence that takes it to state h_S if and only if the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent up to labelling and end in γ_1 and γ_2 respectively.*

PROOF. Observe that the inputs in σ can arrive in an order that alternates between inputs at port 1 and inputs at port 2. However, by construction in this case σ takes $CheckSeq$ to state h_S if and only if $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent up to labelling and end in γ_1 and γ_2 respectively. Thus, if $\sigma \in \mathcal{A}ct^*$ is guaranteed to take $CheckSeq$ to state h_S then the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent up to labelling and end in γ_1 and γ_2 respectively.

Now consider the case where the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent up to labelling and end in γ_1 and γ_2 respectively and we are required to prove that σ must take $CheckSeq$ to state h_S. If the values alternate between ports 1 and 2 then the result follows immediately. We therefore consider the case where the values do not alternate between ports 1 and 2 and so at some point we must have a subsequence of the form $a_p b_p c_q$ for ports $p \neq q$, $a_p, b_p \in \mathcal{A}ct_p$ and $c_q \in \mathcal{A}ct_q$. By construction, if $CheckSeq$ was not already in state 3 then $a_p b_p$ takes it to state $\perp_p$ and c_q then takes $CheckSeq$ to state 3. From state 3 there is no change in state until γ_1 and γ_2 have been received and then $CheckSeq$ moves to state h_S as required. $\Box$

The final component $CheckConf$ checks that two consecutive configurations form a valid change of configuration of $\mathcal{T}M$. This component maintains a flag $conf$ that can take the values 0, 1, 2, or $\perp$. Similar to seq, $conf$ is used to state whether it is known that the current positions in the projections of σ received so far either represent the same configuration ($conf = 0$), the configuration at port 1 represents the one after the configuration at port 2 ($conf = 1$), or the configuration at port 2 represents the one after the configuration at port 1 ($conf = 2$). Once the number of configurations differs by more than one, $conf$ becomes $\perp$. The value of $conf$ is updated when a new configuration separator ($\#_1$ or $\#_2$) is received. If $conf$ is either 1 or 2 and the most recently received values at the ports are $\#_1$ and $\#_2$ then the component checks - as far as possible - that the change in configuration is one that is allowed by $\mathcal{T}M$. This process is only conclusive if the inputs at the two ports, for these two configurations, alternate: as discussed above we can then check in bounded space that the change in configuration is one allowed by $\mathcal{T}M$. If the check is completed and the change in configuration is not one allowed then the component moves to state $\perp_C$ and it cannot then change state.

If this component is not in state $\perp_C$ when γ_1 and γ_2 have been received then it moves to the special state h_C required in h_M.

If $CheckSeq$ is guaranteed to end in state h_S then the projections of σ on ports 1 and 2 must be equivalent (Lemma 6.3). For such a σ, if there are consecutive configurations c and c' such that $\mathcal{T}M$ cannot move from c to c' then there is some $\sigma' \sim \sigma$ that will lead to component $CheckConf$ moving to state $\perp_C$; otherwise, $CheckConf$ cannot move to state $\perp_C$.

LEMMA 6.4. *An input sequence $\sigma \in \mathcal{A}ct^*$ sent by a centralised tester is guaranteed to lead to $CheckSeq \parallel CheckConf$ receiving an input sequence that takes it to state (h_S, h_C) if and only if the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent identical computations of $\mathcal{TM}$.*

PROOF. First assume that trace $\sigma \in \mathcal{A}ct^*$ is guaranteed to take $CheckSeq \parallel CheckConf$ to state (h_S, h_C). By Proposition 6.3 we know that the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ are equivalent.

Now consider the interleaving in which the representation of the first configuration at port 1 arrives and then inputs alternate between the ports. By construction in this case σ takes $CheckSeq \parallel CheckConf$ to state (h_S, h_C) if and only if the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent identical computations of $\mathcal{TM}$. Thus, if $\sigma \in \mathcal{A}ct^*$ is guaranteed to take $CheckSeq \parallel CheckConf$ to state (h_S, h_C) then the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent identical computations of $\mathcal{TM}$.

Now consider the case where the local traces $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent identical computations of $\mathcal{TM}$ and we are required to prove that σ must take $CheckSeq \parallel CheckConf$ to state (h_S, h_C). We will use proof by contradiction and assume that σ is not guaranteed to take $CheckSeq \parallel CheckConf$ to state (h_S, h_C). But this must mean that $CheckConf$ can reach state $\perp_C$ (since otherwise the final γ_1 and γ_2 take it to h_C and we know that $CheckSeq$ must end in h_S). By construction, this can only be the case if $\pi_p(\gamma)$ represents a common sequence $c_1, \ldots, c_k$ of configurations of $\mathcal{TM}$ where there is some $1 \le i < k$ such that $\mathcal{TM}$ cannot move from c_i to c_{i+1}. But this contradicts $\pi_1(\sigma)$ and $\pi_2(\sigma)$ represent identical computations of $\mathcal{TM}$ as required. The result thus follows. □

We are now in the position to prove that it is undecidable whether there is a test case that is guaranteed to move an IOTS model into a particular state. Since such problems are generally undecidable for single port systems if models are allowed to be infinite, we restrict attention to IOTSs with finite sets of states, inputs and outputs.

THEOREM 6.5. *Let us suppose that M is an IOTS with finite sets of states, inputs and outputs. The following problem is generally undecidable even if M has only two ports: given state s of M, is there a test case that is guaranteed to take M to s when testing is centralised and asynchronous?*

PROOF. Given a Turing machine $\mathcal{TM}$ we construct $M_{\mathcal{TM}} = Start_1 \parallel Start_2 \parallel End_1 \parallel End_2 \parallel CheckSeq \parallel CheckConf$ and set $h_M = (s_1, s_2, e_1, e_2, h_S, h_C)$. Consider an input sequence σ that is guaranteed to reach h_M in $M_{\mathcal{TM}}$. From Lemma 6.3 we know that $\pi_1(\sigma) = \sigma_1\gamma_1$ and $\pi_2(\sigma) = \sigma_2\gamma_2$ for some $\sigma_1 \in (\mathcal{A}ct_1 \setminus \{\gamma_1\})^*$ and $\sigma_2 \in (\mathcal{A}ct_2 \setminus \{\gamma_2\})^*$. From Lemma 6.1 we must have that σ_1 and σ_2 both start with copies of the initial configuration of $\mathcal{TM}$ and by Lemma 6.2 we know that σ_1 and σ_2 both end with copies of the halting configuration of $\mathcal{TM}$. By Lemma 6.3 we know that σ_1 and σ_2 are identical up to labelling. Further, by Lemma 6.4 we know that σ_1 and σ_2 represent computations of $\mathcal{TM}$. Thus, σ is guaranteed to take $M_{\mathcal{TM}}$ to h_M if and only if σ_1 and σ_2 represent the same halting computation of $\mathcal{TM}$. The result thus follows from the halting problem for Turing machines being undecidable. □

Now consider the problem of finding a test case that is guaranteed to distinguish two states of an IOTS M. A test case distinguishes two states s and s' of M if there is no observation that can be made when applying the test case when M is in state s and also when applying the test case when M is in state s'. Similar to [Alur et al. 1995; Hierons 2010], we show that the problem of reaching a state can be expressed in terms of distinguishing two states; it is straightforward to show that the result is not affected by whether one can observe quiescence in testing.

THEOREM 6.6. *The following problem is generally undecidable: given states s and s' of IOTS M, is there a test case that is guaranteed to distinguish between s and s' when testing is centralised and asynchronous?*

PROOF. From Theorem 6.5 we know that reachability is undecidable even if we restrict to IOTSs that have no transitions labelled with outputs. We therefore let $M = (S, I, O, T, s_0)$ be an IOTS that has no transitions labelled with outputs ($O = \emptyset$), let $S = \{s_1, \ldots, s_m\}$ and let s_k be a state of M. We now define an IOTS M' where a test case is guaranteed to reach s_k in M if and only if it is guaranteed to distinguish between two particular states of M'. We define two copies M_1 and M_2 of M, where M_1 is a copy of M and M_2 is a copy of M in which we add a transition with label $!o$ from state s_k to a new state s_e from which all transitions are self-loops with input. Then, consider IOTS M' that is the disjoint union of M_1 and M_2 and the problem of distinguishing the initial states of M_1 and M_2. Clearly, a test case is guaranteed to distinguish the initial states of M_1 and M_2 in M' if and only if it is guaranteed to take M to state s_k. Thus, the result follows from Theorem 6.5. $\square$

Since all states except s_k are quiescent it is straightforward to see that the ability to observe quiescence does not affect the proof.

7. CONCLUSIONS

This paper considered the testing of a system that interacts with its environment through distributed interfaces, called ports. Such systems are relatively common, with examples including communications protocols, web services, cloud systems, and wireless sensor networks. For such systems it is not possible to have a single tester that interacts synchronously with all of the ports of the system under test (SUT). Two alternative approaches to testing have previously been discussed: either we have multiple independent distributed testers that interact synchronously with the ports of the SUT or we have a centralised tester that interacts asynchronously with these ports. The former approach has been represented in terms of an implementation relation **dioco** and in this paper we defined an implementation relation $\mathbf{dioco}_c$ for the centralised approach.

We showed that the implementation relations **dioco** and $\mathbf{dioco}_c$ are incomparable and so the two approaches to testing find different traces not defined by the specification and so potentially different faults. Based on this observation we defined a hybrid framework that has a centralised tester and also a local tester at each port with the local tester at port p observing the sequence of events at p. This leads to a strictly stronger implementation relation $\mathbf{dioco}_s$ and so to more effective testing. There are two versions of each implementation relation: one for situations in which quiescence can be observed during testing and one for when it cannot. However, the main results hold for both versions. Interestingly, the hybrid framework is more effective than separately applying both the centralised and distributed approaches.

Having defined the hybrid framework, we explored properties of this and the centralised approach. We proved that the Oracle problem, of deciding whether an observation is consistent with the specification, is NP-complete for both. However, we gave a polynomial time solution for the case where there is an upper bound on the number of ports. This suggests that the Oracle problem being NP-complete should not be a barrier for systems where there are only a few ports. An example of this is communications protocols, which have two ports. It also suggests that there are benefits in using tests that have relatively few local testers. We also showed that in both the centralised and hybrid frameworks it is generally undecidable whether there is a test case that is guaranteed to take an IOTS model to a particular state or that is guaranteed to distinguish two states. The problems considered are all motivated by test automation: either the desire to automatically check an observation against a specification or to automatically generate test cases that achieve certain objectives. The results also have the potential to feed into notions of testability for distributed testing.

It is possible to change the hybrid framework by making one of the local testers also act as the centralised tester. This would lead to a slightly different implementation relation. We did not do this since it would lead to a slightly more involved definition of the $\mathbf{delay}_{\mathcal{P}}$ operator and so would complicate the exposition. However, it should be straightforward to adapt the proofs to such a case by, for example, taking the constructions/IOTSs defined and adding a local tester that is also the centralised tester but makes no local observation. Thus, such a change should not affect the main results.

There are several lines of future work. First, while we know that it is generally undecidable whether there is a test case that is guaranteed to force an IOTS into a particular state or to distinguish two states, it would be interesting to explore conditions under which these problems become decidable. There may also be other useful conditions under which the Oracle problem can be solved in polynomial time. Both of these have the potential to feed into notions of testability for distributed testing. There is also the important problem of extending this work to non-FIFO channels. It is straightforward to adapt the implementation relations and hybrid framework to the case where there are non-FIFO channels; it is sufficient to change the definition of the generalised delay operator to allow inputs to port p to overtake one another and outputs from p also to overtake one another. The proof of the Oracle problem being NP-complete also applies to the non-FIFO case since the trace used in the proof that the problem is NP-hard has only one event at a port p (so the FIFO and non-FIFO cases coincide). In contrast, it appears not to be possible to directly extend the proof that reachability is undecidable.

REFERENCES

F. Aarts and F. W. Vaandrager. 2010. Learning I/O Automata. In *21th International Conference on Concurrency Theory (CONCUR 2010) (Lecture Notes in Computer Science)*, Vol. 6269. Springer, 71–85.

R. Alur, C. Courcoubetis, and M. Yannakakis. 1995. Distinguishing tests for nondeterministic and probabilistic machines. In *27th ACM Symposium on Theory of Computing*. 363–372.

E. Brinksma, L. Heerink, and J. Tretmans. 1998. Factorized Test Generation for Multi-Input/Output Transition Systems. In *11th IFIP International Workshop on Testing Communicating Systems (IWTCS) (IFIP Conference Proceedings)*, Vol. 131. Kluwer, 67–82.

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

J. Chen, R. M. Hierons, and H. Ural. 2005. Resolving Observability Problems in Distributed Test Architectures. In *Formal Techniques for Networked and Distributed Systems (FORTE 2005) (Lecture Notes in Computer Science)*, Vol. 3731. Springer, 219–232.

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

T. S. Chow. 1978. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering* 4 (1978), 178–187.

M. D. Davis and E. J. Weyuker. 1993. *Computability, Complexity and Languages*. Academic Press.

E. D. Demaine and R. A. Hearn. 2008. Constraint Logic: A Uniform Framework for Modeling Computation as Games. In *23rd Annual IEEE Conference on Computational Complexity (CCC 2008)*. 149–162.

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

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

A. En-Nouaary. 2013. A test purpose-based approach for testing timed input output automata. *Journal of Software Testing, Verification and Reliability* 23, 1 (2013), 53–76.

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

R. W. Floyd. 1964. New Proofs and Old Theorems in Logic and Formal Linguistics. Computer Associated Inc, Wakefield, Mas. (1964).

M.-C. Gaudel. 1995. Testing can be formal Too. In *6th International Joint Conference CAAP/FASE Theory and Practice of Software Development (TAPSOFT'95) (Lecture Notes in Computer Science)*, Vol. 915. Springer, 82–96.

G. Gonenc. 1970. A method for the design of fault detection experiments. *IEEE Trans. Comput.* 19 (1970), 551–558.

W. Grieskamp. 2006. Multi-paradigmatic Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification (FATES/RV 2006) (Lecture Notes in Computer Science)*, Vol. 4262. Springer, 1–19.

W. Grieskamp, N. Kicillof, K. Stobie, and V. A. Braberman. 2011. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability* 21, 1 (2011), 55–71.

L. Heerink and J. Tretmans. 1997. Refusal Testing for Classes of Transition Systems with Inputs and Outputs. In *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII) (IFIP Conference Proceedings)*, Vol. 107. Chapman & Hall, 23–38.

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

R. M. Hierons. 2010. Reaching and Distinguishing States of Distributed Systems. *SIAM J. Comput.* 39, 8 (2010), 3480–3500.

R. M. Hierons. 2012a. The complexity of asynchronous model based testing. *Theoretical Computer Science* 451 (2012), 70–82.

R. M. Hierons. 2012b. Oracles for Distributed Testing. *IEEE Transactions on Software Engineering* 38, 3 (2012), 629–641.

R. M. Hierons. 2013. Implementation Relations for Testing Through Asynchronous Channels. *Comput. J.* 56, 11 (2013), 1305–1319.

R. M. Hierons, M. G. Merayo, and M. Núñez. 2012. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing* 25, 1 (2012), 35–62.

R. M. Hierons and H. Ural. 2008a. Checking sequences for distributed test architectures. *Distributed Computing* 21, 3 (2008), 223–238.

R. M. Hierons and H. Ural. 2008b. The Effect of the Distributed Test Architecture on the Power of Testing. *Comput. J.* 51, 4 (2008), 497–510.

J. Huo and A. Petrenko. 2004. On Testing Partially Specified IOTS through Lossless Queues. In *16th IFIP International Conference on the Testing of Communicating Systems (TestCom 2004) (Lecture Notes in Computer Science)*, Vol. 2978. Springer, 76–94.

I. Hwang, A. R. Cavalli, M. Lallali, and D. Verchère. 2012. Applying formal methods to PCEP: an industrial case study from modeling to test generation. *Software Testing, Verification and Reliability* 22, 5 (2012), 343–361.

ISO/IEC. 1995. *Information technology - Opens Systems Interconnection, 9646 Parts 1-7*. ISO/IEC.

C. Jard, T. Jéron, H. Kahlouche, and C. Viho. 1998. Towards Automatic Distribution of Testers for Distributed Conformance Testing. In *TC6 WG6.1 Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE 1998) (IFIP Conference Proceedings)*, Vol. 135. Kluwer, 353–368.

G.-V. Jourdan, H. Ural, and H. Yenigün. 2006. Minimizing Coordination Channels in Distributed Testing. In *Formal Techniques for Networked and Distributed Systems (FORTE 2006) (Lecture Notes in Computer Science)*, Vol. 4229. Springer, 451–466.

D. Lee and M. Yannakakis. 1994. Testing Finite-State Machines: State Identification and Verification. *IEEE Trans. Comput.* 43, 3 (1994), 306–320.

D. Lee and M. Yannakakis. 1996. Principles and Methods of Testing Finite-State Machines - A Survey. *Proc. IEEE* 84, 8 (1996), 1089–1123.

Z. Li, J. Wu, and X. Yin. 2004. Testing Multi Input/Output Transition System with All-Observer. In *16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004) (Lecture Notes in Computer Science)*, Vol. 2978. Springer, 95–111.

T. Miller and P. A. Strooper. 2012. A case study in model-based testing of specifications and implementations. *Software Testing, Verification and Reliability* 22, 1 (2012), 33–63.

E. F. Moore. 1956. Gedanken-Experiments. In *Automata Studies*, C. Shannon and J. McCarthy (Eds.). Princeton University Press.

N. Noroozi, R. Khosravi, M. R. Mousavi, and T. A. C. Willemse. 2011. Synchronizing Asynchronous Conformance Testing. In *9th International Conference on Software Engineering and Formal Methods (SEFM 2011) (Lecture Notes in Computer Science)*, Vol. 7041. Springer, 334–349.

G. L. Peterson and J. H. Reif. 1979. Multiple-Person Alternation. In *20th Annual Symposium on Foundations of Computer Science (FOCS 79)*. IEEE, 348–363.

A. Petrenko, S. Boroday, and R. Groz. 2004. Confirming Configurations in EFSM Testing. *IEEE Transactions on Software Engineering* 30, 1 (2004), 29–42.

A. Petrenko, N. Yevtushenko, and J. Huo. 2003. Testing Transition Systems with Input and Output Testers. In *15th IFIP International Conference on Testing of Communicating Systems (TestCom 2003) (Lecture Notes in Computer Science)*, Vol. 2644. Springer, 129–145.

O. Rafiq and L. Cacciari. 2003. Coordination Algorithm for Distributed Testing. *The Journal of Supercomputing* 24, 2 (2003), 203–211.

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

T. J. Schaefer. 1978. The Complexity of Satisfiability Problems. In *Tenth Annual ACM Symposium on Theory of Computing (STOC)*. 216–226.

A. Simão and A. Petrenko. 2011. Generating asynchronous test cases from test purposes. *Information and Software Technology* 53 (2011), 1252–1262.

J. Swigart and S. Lall. 2011. Optimal controller synthesis for a decentralized two-player system with partial output feedback. In *American Control Conference (ACC), 2011*. 317–323.

L. H. Tahat, B. Korel, M. Harman, and H. Ural. 2012. Regression test suite prioritization using system models. *Software Testing, Verification and Reliability* 22, 7 (2012), 481–506.

K.-C. Tai and Y.-C. Young. 1998. Synchronizable test sequences of finite state machines. *Computer Networks and ISDN Systems* 30, 12 (1998), 1111–1134.

R. E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

J. Tretmans. 1996. Conformance testing with labelled transitions systems: Implementation relations and test generation. *Computer Networks and ISDN Systems* 29, 1 (1996), 49–79.

J. Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing (Lecture Notes in Computer Science)*, Vol. 4949. Springer, 1–38.

M. Veanes and N. Bjørner. 2010. Alternating Simulation and IOCO. In *22nd IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS) (Lecture Notes in Computer Science)*, Vol. 6435. Springer, 47–62.

M. Weiglhofer and F. Wotawa. 2009. Asynchronous Input-Output Conformance Testing. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*. IEEE Computer Society, 154–159.

T.C. Yang, J.H. Zhang, and H. Yu. 1999. A new decentralised controller design method with application to power-system stabiliser design. *Control Engineering Practice* 7, 4 (1999), 537 – 545. DOI:http://dx.doi.org/10.1016/S0967-0661(99)00014-3