Conformance Relations for Distributed Testing Based on CSP

Ana Cavalcanti, Marie-Claude Gaudel, Robert Hierons

Conformance Relations for Distributed Testing based on CSP

Ana Cavalcanti[1], Marie-Claude Gaudel[2], and Robert M. Hierons[3]

[1] University of York, UK
[2] LRI, Université de Paris-Sud and CNRS, France
[3] Brunel University, UK

Abstract. CSP is a well established process algebra that provides comprehensive theoretical and practical support for refinement-based design and verification of systems. Recently, a testing theory for CSP has also been presented. In this paper, we explore the problem of testing from a CSP specification when observations are made by a set of distributed testers. We build on previous work on input-output transition systems, but the use of CSP leads to significant differences, since some of its conformance (refinement) relations consider failures as well as traces. In addition, we allow events to be observed by more than one tester. We show how the CSP notions of refinement can be adapted to distributed testing. We consider two contexts: when the testers are entirely independent and when they can cooperate. Finally, we give some preliminary results on test-case generation and the use of coordination messages.

1 Introduction

As a notation for refinement, CSP has well understood models and associated model-checking techniques and tools [14]. Testing using CSP specifications, however, has not been widely studied yet. In [3], Cavalcanti and Gaudel present a CSP framework for testing against refinement, with a unique tester that has visibility of all interactions with the system under test. In this paper, we investigate the effect of having distributed testers with limited or no global observation.

Distributed and shared systems are increasingly common, but are difficult to observe and control globally; this raises difficulties for testing them. Here, we address these issues for testing based on CSP, in the line of works by Jard et al. [12, 13, 11], Ural and Williams [16], and Hierons and Nunez [7, 8].

To formalise the fact that multiple independent users have a weaker power of observation than a centralised user, it is necessary to state adequate weaker notions of refinement, as proposed for CSP by Jacob [10], or similarly weaker conformance relations, as developed as alternatives for the well-known *ioco* relation in [7, 9]. This paper studies such refinements relations for CSP.

First, we consider cooperating refinement, where there is a possibility of collectively checking the observations at some points (namely, after complete runs). Second, we study independent refinement, where there is no way for synthesizing observations. The notions of cooperating and independent refinement have

been introduced in [10] for a general unspecified notion of observation. Here, we instantiate these relations to obtain CSP versions of the conformance relations studied for Input/Output Transition Systems (IOTSs) in [7, 8].

We relate the notion of independent refinement to that of lazy abstraction in [14]. In this way, we reveal the natural instantiation of independent refinement when the observations are failures; this is not covered in [7, 8]. Via a number of examples, we explore the properties of the relations introduced here. Finally, we briefly consider test generation. The approach previously devised for CSP [3] can be adapted, but the resulting test cases need not be sound. We then show how the use of coordination messages suggested by Hierons in [5] can be adapted to CSP to produce sound test cases that establish traces refinement.

The paper is organised as follows. In the next section, we give an overview of CSP and the existing work on distributed testing for IOTSs. Section 3 introduces and discusses our proposed definitions of cooperating, and independent traces and failures refinement. In Section 4, we consider coordination messages. We draw our conclusions, and discuss future work in our final Section 5.

2 Preliminaries

We cover aspects of CSP, and relevant results on distributed testing for IOTSs.

2.1 CSP

In CSP, systems (and their components) are modelled as processes that interact synchronously with each other and their environment via events representing communications over channels. The set of (external) events in which a process P can engage is denoted αP. Sets of events are called alphabets.

The process $STOP$ is deadlocked, $SKIP$ terminates immediately, and $a \rightarrow P$ can engage in the event a, and then behave like the process P. An external choice $P_1 \,\Box\, P_2$ offers to its environment the behaviour of either P_1 or P_2; the choice is determined by the first event on which the environment synchronises. An internal choice $P_1 \sqcap P_2$ is nondeterministic; it can behave as either P_1 or P_2.

Processes can also be combined in parallel. We use the alphabetised parallelism: $P_1 \,[\![\, A \,]\!]\, P_2$, which executes P_1 and P_2 concurrently, requiring that they synchronise on the events in the set A. We also use the replicated parallel operator $\|\ i : I \bullet [A(i)]P(i)$, where the processes $P(i)$ with alphabet $A(i)$, for i in the indexing set I, are run in parallel, synchronising on their common channels.

Events can be external, that is, observable and controllable by the environment, or internal. Using the hiding operator, like in $P \setminus A$, we define a process that behaves like P, but whose events in the set A are internal.

CSP has three standard semantic models: the traces, the (stable) failures, and the failures-divergences models. In the traces model, a process P is characterised by its set $traces(P)$ of traces t of type $\operatorname{seq} \Sigma^{\checkmark}$. These are finite sequences of events in which it can engage. The special event $\checkmark$ records termination. The empty trace is $\langle\,\rangle$. The set of all events, excluding $\checkmark$, is Σ; the set with $\checkmark$ is $\Sigma^{\checkmark}$.

The set $traces(P)$, for process P, is prefix closed: if a process can engage in a given sequence of events, then it can engage in all its prefixes.

For a trace t of a process P and a subset R of $\Sigma^{\checkmark}$, the pair (t, R) is a failure for P if, and only if, after performing t, P may refuse all events of R. The set $failures(P)$ containing all failures of P is subset closed: if P may deadlock when the choice among events in a set R is proposed by its environment after a trace t, it may deadlock as well if only a subset of R is proposed.

The set $divergences(P)$ contains the traces of P that lead to an infinite sequence of internal events. The canonical semantics of CSP is based on its failures-divergences model $\mathcal{N}$, where the set of traces is determined by the set of *failures*. There are also CSP models that record the set $infinites(P)$ of infinite traces of P. They capture unbounded nondeterminism more accurately [14].

As usual, we assume that specifications and systems are divergence free. A divergent specification is necessarily a mistake. Also, when testing, divergences raise problems of observability. Therefore, we identify divergence with deadlock.

In the traces model, a process P is defined to be trace refined by a process Q, that is, $P \sqsubseteq_T Q$, if, and only if, $traces(Q) \subseteq traces(P)$. For divergence-free processes, the notion of refinement in the canonical model $\mathcal{N}$ of CSP is failures refinement $P \sqsubseteq_F Q$, which requires $failures(Q) \subseteq failures(P)$. For the model of infinite traces, we consider $P \sqsubseteq_\infty Q$, which, when P and Q are divergence-free, also requires reverse containment of (both finite and infinite) traces.

All these models and notions of refinement are based on the possibility of global observations of the system. Later, in Section 3, we consider distribution.

2.2 Distributed testing for IOTS

Most work on formal distributed testing concerns testing from a Deterministic Finite State Machines (DFSM). While DFSMs are suitable for specifying some classes of systems, they require that the set of states is finite and that input and output alternate. In addition, many distributed systems are nondeterministic. There has been, thus, interest in distributed testing from an IOTS [1].

In this paper we build on recent work that defines conformance relations for distributed testing from an IOTS [7, 6]. It considers two scenarios. In the first, the testers are independent in that no external agent can receive information from more than one of them. Here, it is sufficient that the local behaviour observed by a tester is consistent with a trace of the specification.

The implementation relation **p-dioco** is based on this idea; it requires that for each finite trace σ of the implementation and tester p there is a trace σ' of the specification such that the projections of σ and σ' at p are identical. An important characteristic of **p-dioco** is that given a trace σ of the implementation, the trace σ' that the specification uses to simulate it can vary with the tester.

In the second scenario, there is the possibility that information from two or more testers can be received by an external agent. As a result, the local behaviours observed by the testers could be brought together and so a stronger implementation relation **dioco** is introduced.

Work on DFSM has identified the controllability problem, which occurs when a test leads to a situation where a tester does not know when to apply an input [15]. As an example, we consider a test that starts with an input $?i_1$ that is to be applied by a tester 1 and lead to an output $!o_1$ for 1, after which the input $?i_2$ should be applied by a tester 2. The problem is that 2 does not observe $?i_1$ or $!o_1$, and so does not know when to apply $?i_2$.

The reduction in observational power also affects the ability of distinguishing between a trace of the specification and a trace of the implementation during testing. This has been called an observability problem [4].

Generating tests without controllability problems restricts testing. An alternative is to overcome these problems through the exchange of coordination messages between testers [2]. It has been shown that, when testing from an IOTS, coordination messages can be used to allow global traces to be observed, but this requires several coordination messages for each event [12]. Recent work has shown how fewer messages can be added to a test sequence [5] to overcome controllability problems. It is this approach that we adapt.

3 Distributed testing for CSP

In this section, we define for CSP relations corresponding to **dioco** and **p-dioco**; more precisely, we define notions of cooperating and independent refinement. We consider observations of both traces and failures, but not divergences.

Our work builds on that presented in [10], which considers notions of refinement for CSP processes, when the environment consists of several users. It proposes general notions of cooperating and independent refinement that we instantiate here for the observations of interest in the scenarios studied in [7]. In [10] they are used to define traces-based refinement for transactions.

We characterise users U by non-empty sets of events. Unlike [10], we do not assume that these sets are disjoint, and actually require that $\checkmark$ can be observed by all users. Additionally, to allow the use of synchronous coordination messages in testing experiments, users need to have non-disjoint alphabets. We use $\mathcal{A}$ to denote the finite set of all users and assume that $\bigcup \mathcal{A} = \Sigma^{\checkmark}$. In examples we do not explicitly list the event $\checkmark$ when defining a user, since it is always included.

3.1 Cooperating refinement

Like **dioco**, cooperating refinement caters for a scenario in which the observations of the various users are reconciled at some point. This means that the users can compare their observations, and what is checked is that, collectively, their observations can account for one single behaviour of the process.

If the users get together too soon, or compare their observation at different stages of the interaction, then inappropriate distinctions can be made.

Example 1. We consider the specification $P = a \rightarrow b \rightarrow STOP$ and users $\{a\}$ and $\{b\}$. If we have an accurate implementation, $\{a\}$ observes the traces $\langle\,\rangle$ and

$\langle a \rangle$. The traces for $\{b\}$ are $\langle\,\rangle$ and $\langle b \rangle$. If $\{a\}$ observes $\langle a \rangle$, and $\{b\}$ observes $\langle\,\rangle$, then in comparing their observations we have the global trace $\langle a \rangle$. We cannot, however, compare all pairs of traces observed by $\{a\}$ and $\{b\}$. There is, for instance, no global trace corresponding to $\langle\,\rangle$ from $\{\,a\,\}$, and $\langle b \rangle$ from $\{b\}$. □

Complete runs There are various ways of identifying the observations that are of interest for comparison. Here, we pursue the solution proposed by the **dioco** relation, which is based on the notion of a complete run.

For CSP, we define the set $\mathcal{C}(P)$ of complete runs of a process P in the infinite-traces model. Namely, the complete runs are those characterised by traces that record a termination, lead to a deadlock, or are infinite. The need for infinite traces is justified, for instance, by the process $P = a \rightarrow P$, which does not terminate or deadlock. If we consider the model $\mathcal{N}$, P has no complete runs.

Definition 1 (Complete run).

$$\mathcal{C}(P) \mathrel{\widehat{=}} \{\, t : \mathit{traces}(P) \mid \mathrm{last}\, t = \checkmark \vee (t, \Sigma^{\checkmark}) \in \mathit{failures}(P) \,\} \cup \mathit{infinites}(P)$$

For any finite sequence s, we use $\mathrm{last}\, s$ to denote its last element.

Local equivalence Cooperating refinement is based on a notion of local equivalence for traces. For traces s and t, we write $s \sim_T t$ if s and t are locally trace equivalent (with respect to the set of users $\mathcal{A}$). This means that the set of individual observations of the users in $\mathcal{A}$ cannot distinguish s from t.

Definition 2 (Local trace equivalence).

$$s \sim_T t \mathrel{\widehat{=}} (\forall\, U : \mathcal{A} \bullet \pi_U(s) = \pi_U(t))$$

where, for every trace t and user U, $\pi_U(t) = t \upharpoonright U$.

The sequence $s \upharpoonright F$ is that obtained from s by removing all elements not in F.

It does not make sense to define a similar failure-based equivalence, since we only consider complete runs. All sets of events are refused after termination or a deadlock, and there are no failures for infinite traces.

Definition and properties Using the notion of complete run, we define cooperating traces refinement as a direct instantiation of the definition in [10].

Definition 3 (Cooperating traces refinement).

$$P \sqsubseteq_{CT} Q \mathrel{\widehat{=}} \forall\, s : \mathcal{C}(Q) \bullet \exists\, t : \mathcal{C}(P) \bullet s \sim_T t$$

A process P is cooperating refined by Q if, for every complete run of Q there is a complete run of P that is local trace equivalent to it.

Example 2. The only complete run of $P = a \rightarrow b \rightarrow \mathit{SKIP}$ is $\langle a, b, \checkmark \rangle$. The complete runs of $Q = a \rightarrow b \rightarrow \mathit{SKIP} \;\Box\; b \rightarrow a \rightarrow \mathit{SKIP}$ are $\langle a, b, \checkmark \rangle$ and $\langle b, a, \checkmark \rangle$. If we consider users $\{a\}$ and $\{b\}$, then $\langle a, b, \checkmark \rangle$ is locally equivalent to $\langle b, a, \checkmark \rangle$. Therefore, not only $Q \sqsubseteq_{CT} P$, but also $P \sqsubseteq_{CT} Q$. In other words, P and Q are equal from the point of view of cooperating refinement. This reflects the fact that the users do not have a record of the time in which their observations are made, and so cannot compare their relative order. □

It is not difficult to see that, in general, $P \sqsubseteq_\infty Q$ implies $P \sqsubseteq_{CT} Q$, since in this case all observations (traces, including the infinite ones, and failures) of Q are also observations of P. This, of course, includes the complete runs of Q.

Traces refinement, however, does not entail cooperating refinement.

Example 3. The processes $P = a \rightarrow STOP \sqcap STOP$ and $Q = a \rightarrow STOP$ are equal from the point of view of traces refinement. On the other hand, $\mathcal{C}(P) = \{\langle\rangle, \langle a\rangle\}$ and $\mathcal{C}(Q) = \{\langle a\rangle\}$. Since $\langle\rangle$ has no equivalent in $\mathcal{C}(Q)$ for a user $\{a\}$, we have that $P \sqsubseteq_{CT} Q$, but not $Q \sqsubseteq_{CT} P$. □

Cooperating refinement, and all other relations presented here, are not compositional. They are, therefore, in general not amenable to compositional analysis.

Example 4. The processes $P = a \rightarrow b \rightarrow STOP$ and $Q = b \rightarrow a \rightarrow STOP$ are equal (for users $\{a\}$ and $\{b\}$) from the point of view of cooperating refinement. Their complete runs $\mathcal{C}(P) = \{\langle a, b\rangle\}$ and $\mathcal{C}(Q) = \{\langle b, a\rangle\}$ are locally equivalent. We consider, however, the context defined by the process function F below.

$$F(X) = (X \,[\![\, \{a, b\} \,]\!]\, b \rightarrow STOP)$$

We have that $F(P) = STOP$ and $F(Q) = b \rightarrow STOP$. From the point of view of a user $\{b\}$, these processes can be distinguished. □

Lack of compositionality restricts the opportunities of practical (and scalable) use of our relations for development and analysis. For testing of complete systems, however, this is not an issue, and we expect that certain architectural patterns ensure compositionality. This will be considered in our future work.

Since local equivalence is transitive, so is cooperating refinement.

3.2 Independent refinement

The scenario considered in independent refinement is similar to that in **p-dioco**, namely, a situation in which the users do not have a way of comparing their observations. Here, we consider both observations of traces and failures.

Independent traces refinement The **p-dioco** relation is the inspiration for what we call here independent traces refinement, and define as follows.

Definition 4 (Independent traces refinement).

$$P \sqsubseteq_{IT} Q \mathrel{\widehat{=}} (\forall\, U : \mathcal{A};\ s : traces(Q) \bullet (\exists\, t : traces(P) \bullet \pi_U(s) = \pi_U(t)))$$

For every user U and trace s of Q, we require there to be a trace t of P such that U cannot distinguish between s and t. This is different from cooperating traces refinement, where we require the existence of a single corresponding trace t in P that cannot be distinguished from s from the point of view of all users.

Example 5. The processes $P = a.1 \rightarrow b.1 \rightarrow STOP \;\Box\; a.2 \rightarrow b.2 \rightarrow STOP$ and $Q = a.1 \rightarrow b.2 \rightarrow STOP \;\Box\; a.2 \rightarrow b.1 \rightarrow STOP$ are different under cooperating refinement with users $\{a.1, a.2\}$ and $\{b.1, b.2\}$. For instance, the complete run $\langle a.1, b.2\rangle$ of Q is not equivalent to any of the complete runs $\langle a.1, b.1\rangle$ and $\langle a.2, b.2\rangle$ of P. In the case of $\langle a.1, b.1\rangle$, the user $\{b.1, b.2\}$ can make a distinction, and for $\langle a.2, b.2\rangle$, the user $\{a.1, a.2\}$ can detect a distinction.

Under independent (traces) refinement, however, P and Q cannot be distinguished, because there is no opportunity for the users to compare their observations. For example, for the trace $\langle a.1, b.2\rangle$ of Q, we have in P the trace $\langle a.1\rangle$ for the user $\{a.1, a.2\}$, and the trace $\langle a.2, b.2\rangle$ for the user $\{b.1, b.2\}$. $\Box$

The processes P and Q, and process function F in Example 4 also provide an example to show that independent traces refinement is not compositional. Namely $P =_{IT} Q$, but not $F(P) =_{IT} F(Q)$; as before $F(P)$ and $F(Q)$ can be distinguished by $\{b\}$, with no trace in $F(P)$ corresponding to $\langle b\rangle$.

Lazy abstraction The notion of independence is related to that of lazy abstraction [14, page 297]. In that work, it is discussed under the assumption that the processes are divergence-free and nonterminating, that the nondeterminism is bounded, and that users are disjoint. In what follows, we establish the relationship between lazy abstraction and independent traces refinement. Afterwards, we use that as inspiration to define independent failures refinement.

Following [14], we define the process $P @ U$, which characterises the behaviour of P for a user U. Below, we define the traces and stable failures of $P @ U$.

In considering the independent behaviour of P from the point of view of U, we observe that the behaviour of other users might affect the perception that U has of P. First of all, there is the possibility of the introduction of deadlock. If, for example, U is waiting for an event b that is only available after P engages in an event a that is not under the control of U, then U may experience a deadlock. This is because the users that control a may not agree on that event.

A second aspect is related to divergences. Like in [14], we assume that divergence is not introduced, even if P offers an infinite trace of events of a user different from U, and therefore an infinite trace of events effectively hidden from U. This means that we assume that no user is fast enough to block P, or that P is fair. As a consequence, what we reproduce below is the canonical failures-divergences model of $P @ U$ if P, and therefore, $P @ U$ are divergence-free [14].

Definition 5 ($P @ U$).

$$\mathit{traces}(P @ U) \mathrel{\widehat=} \{\, t : \mathit{traces}(P) \bullet \pi_U(t) \,\}$$
$$\mathit{failures}(P @ U) \mathrel{\widehat=} \{\, t : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\, \Sigma^{\checkmark} \mid (t, A \cap U) \in \mathit{failures}(P) \bullet (\pi_U(t), A) \,\}$$

The set $\mathit{traces}(P @ U)$ contains the traces $\pi_U(t)$ obtained by removing from a trace t of P all events not in U. The alphabet of refusals, on the other hand, is $\Sigma^{\checkmark}$. (This allows us to compare the views of the different users, and the view of a user with that of the system.) Therefore, the failures $(\pi_U(t), A)$ of $P @ U$ are obtained by considering the failures $(t, A \cap U)$ of P. For the trace t, we consider

$\pi_U(t)$, as already explained. For the refusal $A \cap U$, we observe that if $A \cap U$ is refused by P, then A, which can contain events not in U, is refused by $P @ U$. Since an event not in U cannot be observed by U, it is refused by $P @ U$.

Example 6. We consider the process $P = a \rightarrow b \rightarrow STOP$, and a user $\{b\}$. The set $traces(P @ \{b\})$ is $\{\langle\rangle, \langle b\rangle\}$. The traces $\langle a\rangle$ and $\langle a, b\rangle$ of P are not (entirely) visible to $\{b\}$. The failures of $P @ \{b\}$, on the other hand, include $(\langle\,\rangle, \{a, b\})$ (and so, all subsets of $\{a, b\}$). This indicates that, from the point of view of $\{b\}$, the process can deadlock, since interaction with a may not go ahead. □

The following lemma states that independent trace refinement holds when all independent users observe a traces refinement.

Lemma 1. $P \sqsubseteq_{IT} Q \Leftrightarrow \forall\, U : \mathcal{A} \bullet (P @ U) \sqsubseteq_T (Q @ U)$

Proof.

$\forall\, U : \mathcal{A} \bullet (P @ U) \sqsubseteq_T (Q @ U)$

$\Leftrightarrow \forall\, U : \mathcal{A} \bullet traces(Q @ U) \subseteq traces(P @ U)$ [definition of $\sqsubseteq_T$]

$\Leftrightarrow \forall\, U : \mathcal{A} \bullet (\forall\, s : traces(Q @ U) \bullet (\exists\, t : traces(P @ U) \bullet s = t))$ [property of sets]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A} \bullet (\forall\, s : \operatorname{seq} \Sigma^{\checkmark} \mid (\exists\, so : traces(Q) \bullet s = \pi_U(so)) \bullet \\ \quad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark};\ to : traces(P) \bullet t = \pi_U(to) \wedge s = t)) \end{array}\right)$$

[definition of $traces(P @ U)$ and $traces(Q @ U)$]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A} \bullet (\forall\, s : \operatorname{seq} \Sigma^{\checkmark} \mid (\exists\, so : traces(Q) \bullet s = \pi_U(so)) \bullet \\ \quad (\exists\, to : traces(P) \bullet s = \pi_U(to))) \end{array}\right)$$

[one-point rule]

$\Leftrightarrow \forall\, U : \mathcal{A};\ so : traces(Q) \bullet (\exists\, to : traces(P) \bullet \pi_U(so) = \pi_U(to))$

[one-point rule]

$\Leftrightarrow P \sqsubseteq_{IT} Q$ [definition of $P \sqsubseteq_{IT} Q$]

□

It is a straightforward consequence of the above lemma that independent traces refinement is transitive, since traces refinement is transitive.

Example 7. We consider again $P = a.1 \rightarrow b.1 \rightarrow STOP \,\Box\, a.2 \rightarrow b.2 \rightarrow STOP$ and $Q = a.1 \rightarrow b.2 \rightarrow STOP \,\Box\, a.2 \rightarrow b.1 \rightarrow STOP$. So, $traces(P @ \{a.1, a.2\})$ is $\{\langle\rangle, \langle a.1\rangle, \langle a.2\rangle\}$ and $traces(P @ \{b.1, b.2\})$ is $\{\langle\rangle, \langle b.1\rangle, \langle b.2\rangle\}$. These are also the traces of $Q @ \{a.1, a.2\}$ and $Q @ \{b.1, b.2\}$, as expected from our earlier conclusion that P and Q are indistinguishable under independent traces refinement. □

Example 8. The processes $P = a \rightarrow b \rightarrow STOP$ and $Q = b \rightarrow a \rightarrow STOP$ are not related by traces refinement. As we have already seen, they cannot be distinguished under cooperating refinement with users $\{a\}$ and $\{b\}$. It turns out that these processes are also equal under independent traces refinement. This is because $traces(P @ \{a\}) = \{\,\langle\,\rangle, \langle\, a\rangle\,\}$ and $traces(P @ \{b\}) = \{\,\langle\,\rangle, \langle\, b\rangle\,\}$, and the same holds if we consider Q instead of P. This again reflects the fact that, in isolation, $\{a\}$ and $\{b\}$ cannot decide in which order the events occur. □

Independent failures refinement The definition of $P \sqsubseteq_{IT} Q$ is inspired by [10], and it is interesting that it is similar to the definition of **p-dioco** [7]. Lemma 1 indicates the way for considering also independent failures refinement.

Definition 6 (Independent failures refinement).

$$P \sqsubseteq_{IF} Q \mathrel{\widehat{=}} \forall\, U : \mathcal{A} \bullet (P \mathbin{@} U) \sqsubseteq_{F} (Q \mathbin{@} U)$$

Example 9. Independent failures refinement does not hold (in either direction) for the processes P and Q in Example 8. Intuitively, this is because, from the point of view of $\{a\}$, P is immediately available for interaction, and then deadlocks. On the other hand, Q may deadlock immediately, if b does not happen. Similarly, from the point of view of $\{b\}$, P may deadlock immediately, but not Q. Accordingly, the failures of P and Q for these users are as sketched below.

$$\begin{array}{l} \mathit{failures}(P \mathbin{@} \{a\}) = \{\, (\langle\,\rangle, \{b, \checkmark\}), \ldots, (\langle a\rangle, \{a, b, \checkmark\}), \ldots \} \\ \mathit{failures}(P \mathbin{@} \{b\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle b\rangle, \{a, b, \checkmark\}), \ldots \} \\ \mathit{failures}(Q \mathbin{@} \{a\}) = \{\, (\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle a\rangle, \{a, b, \checkmark\}), \ldots \} \\ \mathit{failures}(Q \mathbin{@} \{b\}) = \{\, (\langle\,\rangle, \{a, \checkmark\}), \ldots, (\langle b\rangle, \{a, b, \checkmark\}), \ldots \} \end{array}$$

We omit the failures that can be deduced by the fact that these sets are subset closed. Deadlock is characterised by a failure whose refusal has all events. □

Example 10. For an example where independent failures refinement holds, consider users $\{\, \{a\}, \{b\}, \{c.1, c.2\} \,\}$, $P = a \to c.1 \to \mathit{STOP} \;\Box\; b \to c.2 \to \mathit{STOP}$, and $Q = a \to c.2 \to \mathit{STOP} \;\Box\; b \to c.1 \to \mathit{STOP}$. We have the following.

$$\begin{array}{l} \mathit{traces}(P \mathbin{@} \{a\}) = \mathit{traces}(Q \mathbin{@} \{a\}) = \{\, \langle\,\rangle, \langle a\rangle \,\} \\ \mathit{traces}(P \mathbin{@} \{b\}) = \mathit{traces}(Q \mathbin{@} \{b\}) = \{\, \langle\,\rangle, \langle b\rangle \,\} \\ \mathit{traces}(P \mathbin{@} \{c.1, c.2\}) = \mathit{traces}(Q \mathbin{@} \{c.1, c.2\}) = \{\, \langle\,\rangle, \langle c.1\rangle, \langle c.2\rangle \,\} \end{array}$$

Regarding refusals, for both P and Q, the view of $\{a\}$ is that the process may nondeterministically choose between deadlocking (if b reacts "more quickly" and takes the choice) or doing an a. The situation for $\{b\}$ is similar. Finally, for $\{c.1, c.2\}$, there is a nondeterministic choice between a deadlock, if neither a nor b happens, or carrying out a $c.1$ or a $c.2$ and then deadlocking. Accordingly, the failures obtained from P and Q are the same; they are sketched below.

$$\begin{array}{l} \mathit{failures}(P \mathbin{@} \{a\}) = \mathit{failures}(Q \mathbin{@} \{a\}) = \\ \quad \{\, (\langle\,\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots (\langle a\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \} \\ \mathit{failures}(P \mathbin{@} \{b\}) = \mathit{failures}(Q \mathbin{@} \{b\}) = \\ \quad \{\, (\langle\,\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots (\langle b\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \} \\ \mathit{failures}(P \mathbin{@} \{c.1, c.2\}) = \mathit{failures}(Q \mathbin{@} \{c.1, c.2\}) = \\ \quad \{\, (\langle\,\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \\ \qquad (\langle c.1\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots, (\langle c.2\rangle, \{a, b, c.1, c.2, \checkmark\}), \ldots \} \end{array}$$

This reflects the fact that no user can observe whether the communication on c follows an a or a b. □

Unlike the (standard) failures-refinement relation, independent failures refine-

ment cannot be used to reason about deadlock.

Example 11. The process $P = a \to STOP \,\Box\, b \to STOP$ is independent failures refined by $STOP$ for users $\{a\}$ and $\{b\}$, since for both of them an immediate deadlock is possible. We have the following failures.

$$failures(P@\{a\}) = \{\,(\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle a\rangle, \{a, b, \checkmark\}), \ldots\}$$
$$failures(P@\{b\}) = \{\,(\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle b\rangle, \{a, b, \checkmark\}), \ldots\}$$

The set of failures of $STOP$, for both users, is $\{\,(\langle\,\rangle, \{a, b, \checkmark\}), \ldots\}$, which is a subset of the sets above. So, a deadlocked implementation is correct with respect to P, under independent failures refinement. □

Using the above result, the example below establishes that, like the other relations defined previously, independent failures refinement is not compositional.

Example 12. We define the process function $F(X) = (X \,[\![\{a\}]\!]\, a \to b \to STOP)$. If P is as defined in Example 11, then $F(P) = a \to b \to STOP \,\Box\, b \to STOP$ and $F(STOP) = STOP$. Now, failures of $F(P)@\{b\}$ includes just $(\langle\,\rangle, \emptyset)$ for the empty trace. So, it is not the case that $F(P) \sqsubseteq_{IF} STOP$. □

Additionally, in some cases, internal and external choice may be perceived by individual users in the same way. An example is provided below.

Example 13. Process $P = a \to STOP \,\Box\, b \to STOP$ is equal, under independent failures refinement, to $Q = a \to STOP \sqcap b \to STOP$ if the users are $\{a\}$ and $\{b\}$. This is because, for P or Q, it is possible for $\{a\}$ or $\{b\}$ to observe a deadlock. For $\{a\}$, for instance, in the case of P, deadlock can happen if $\{b\}$ is quicker in making its choice, and in the case of Q, if the internal choice is made in favour of $b \to STOP$. A similar situation arises for the user $\{b\}$. □

This does not mean, however, that internal and external choice are indistinguishable using independent failures refinement.

Example 14. We now consider $P = a \to b \to STOP \,\Box\, b \to a \to STOP$ and $Q = a \to b \to STOP \sqcap b \to a \to STOP$, then we do not have an equality. In the case of P, the user $\{a\}$, for example, never experiences a deadlock, but in the case of Q, if $b \to a \to STOP$ is chosen, then a deadlock may occur for $\{a\}$, if $\{b\}$ is not ready for interaction. Accordingly, we have the following failures.

$$failures(P@\{a\}) = \{\,(\langle\,\rangle, \{b, \checkmark\}), \ldots, (\langle a\rangle\,, \{a, b, \checkmark\}), \ldots\}$$
$$failures(Q@\{a\}) = \{\,(\langle\,\rangle, \{a, b, \checkmark\}), \ldots, (\langle a\rangle, \{a, b, \checkmark\}), \ldots\}$$

With the empty trace, there is no refusal of $P@\{a\}$ including a. □

As already discussed, in CSP, a process is in charge of the internal choices, and the environment, as a user, has no control over how they are made. With multiple users, we have the possibility of introducing more nondeterminism (from the point of view of a particular user), as there are more players who may be in sole control of choices that the process itself leaves to the environment.

The proof of the following is trivial. It considers the standard case in which the environment is a single user U that can observe every event: $\mathcal{A} = \{\Sigma^{\checkmark}\}$.

Lemma 2. $P \circledcirc \Sigma^{\checkmark} = P$

From this result, Lemma 1 and Definition 3, we conclude that independent refinement amounts to the traditional refinement relations if there is a single observer with a global view. Thus, existing exhaustive test sets for CSP apply in this case.

We give below another characterisation of independent failures refinement.

Lemma 3.

$$P \sqsubseteq_{IF} Q \Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ s : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\, \Sigma^{\checkmark} \mid (s, A \cap U) \in \mathit{failures}_{\perp}(Q) \bullet \\ \quad \exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}_{\perp}(P) \wedge \pi_U(s) = \pi_U(t) \end{array} \right)$$

Proof.

$P \sqsubseteq_{IF} Q$

$\Leftrightarrow \forall\, U : \mathcal{A} \bullet (P \circledcirc U) \sqsubseteq_F (Q \circledcirc U)$ [definition of $\sqsubseteq_{IF}$]

$\Leftrightarrow \forall\, U : \mathcal{A} \bullet \mathit{failures}(Q \circledcirc U) \subseteq \mathit{failures}(P \circledcirc U)$ [definition of $\sqsubseteq_F$]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ su : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\, \Sigma^{\checkmark} \mid (su, A) \in \mathit{failures}(Q \circledcirc U) \bullet \\ \quad (su, A) \in \mathit{failures}(P \circledcirc U) \end{array} \right)$$

[property of sets]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ su : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\, \Sigma^{\checkmark} \mid \\ \quad (\exists\, s : \operatorname{seq} \Sigma^{\checkmark} \bullet (s, A \cap U) \in \mathit{failures}(Q) \wedge su = \pi_U(s)) \bullet \\ \qquad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}(P) \wedge su = \pi_U(t)) \end{array} \right)$$

[definition of $\mathit{failures}(Q \circledcirc U)$ and $\mathit{failures}(P \circledcirc U)$]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ su : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\, \Sigma^{\checkmark};\ s : \operatorname{seq} \Sigma^{\checkmark} \mid \\ \quad (s, A \cap U) \in \mathit{failures}(Q) \wedge su = \pi_U(s) \bullet \\ \qquad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}(P) \wedge su = \pi_U(t)) \end{array} \right)$$

[predicate calculus]

$$\Leftrightarrow \left(\begin{array}{l} \forall\, U : \mathcal{A};\ s : \operatorname{seq} \Sigma^{\checkmark};\ A : \mathbb{P}\, \Sigma^{\checkmark} \mid (s, A \cap U) \in \mathit{failures}(Q) \bullet \\ \quad (\exists\, t : \operatorname{seq} \Sigma^{\checkmark} \bullet (t, A \cap U) \in \mathit{failures}(P) \wedge \pi_U(s) = \pi_U(t)) \end{array} \right)$$

[one-point rule]

□

This states that $P \sqsubseteq_{IF} Q$ requires that, for every user U, every failure of Q whose refusal includes only events visible to U, has a corresponding failure in P. The failures can have different traces s and t, as long as they are the same from the point of view of U. The refusals must be the same.

Revisiting divergences To remove the assumption that the behaviour of other users cannot cause user U to experience divergence, we need a different abstraction $P @^d U$ of P for U. If we assume that P cannot terminate, we can use $P @^d U = (P \,[\![\, \overline{U} \,]\!]\, \mathit{Chaos}(\overline{U})) \setminus \overline{U}$, where $\overline{U} = \Sigma \setminus U$ is the set of events under the control of other users. They are hidden in $P @^d U$, where the parallelism captures the fact that the behaviour of other users is arbitrary. Process $\mathit{Chaos}(A) = \mathit{STOP} \sqcap (\Box\, e : A \bullet e \to \mathit{Chaos}(A))$ can deadlock or perform any event in A at any time. Divergence arises in $P @^d U$ if P offers an infinite sequence of events in $\overline{U}$. This abstraction is suggested in [14], but there the introduction of divergence is considered inadequate. In testing, it is best not to make assumptions about the possible behaviours of a user.

In the traces model $P @ U = P @^d U$, so using $P @^d U$ makes no difference for independent traces refinement. Additionally, to take into account the possibility that P may terminate, we need only to ensure that the parallelism in $P @^d U$ terminates when P does. To provide a more general definition that considers terminating P, we can, therefore, consider, for example, a (syntactic) function that changes P to indicate its imminent termination using a fresh event ok. With this modified version $\mathcal{OK}(P)$ of P, to define $P @^d U$ we can use $(\mathcal{OK}(P) \,[\![\, \overline{U} \,]\!]\, (\mathit{Chaos}(\overline{U}) \,\Box\, ok \to \mathit{SKIP})) \setminus (\overline{U} \cup \{ok\})$. The failures of $P @^d U$ include those of $P @ U$ plus those that can arise from divergence.

4 Distributed testing and traces refinement

In this section we discuss distributed testing from CSP.

4.1 Global testing for traces refinement

Since traces refinement $P \sqsubseteq_T Q$ prescribes $\mathit{traces}(Q) \subseteq \mathit{traces}(P)$, but not the reverse, there is no need to test that Q can execute the traces of P. It is sufficient to test Q against those traces of events in the alphabet of P that are not traces of P, and to check that they are refused. Moreover, it is sufficient to consider the minimal prefixes of forbidden traces that are forbidden themselves. Formally, testing for traces refinement is performed by proposing to the system under test the traces $s \frown \langle\, a \,\rangle$, where s is in $\mathit{traces}(P)$, and a is a forbidden continuation.

For one test execution, the verdict is as follows. If s is followed by a deadlock, then the test execution is said to be a *success*. If $s \frown \langle a \rangle$ is observed, the result is a *failure*. If a strict prefix of s followed by a deadlock is observed, then the execution is *inconclusive*. In this case, the trace s of P has not been executed by the system under test. As explained above, according to traces refinement, we do not have a failure, but the test does not produce conclusive information.

In [3], the three special events in the set $V = \{\mathit{pass}, \mathit{fail}, \mathit{inc}\}$ are introduced to perform on-the-fly verdict. For a finite trace $s = a_1, a_2, \ldots, a_n$ and a forbidden continuation event a, the CSP test process $T_T(s, a)$ is defined as follows.

$$T_T(s, a) = \mathit{inc} \to a_1 \to \mathit{inc} \to a_2 \to \mathit{inc} \ldots a_n \to \mathit{pass} \to a \to \mathit{fail} \to \mathit{STOP}$$

As explained above, the last event before a deadlock gives the verdict.

Formally, we can define $T_T(s, a)$ inductively as shown below.

$$T_T(\langle\,\rangle, a) = pass \rightarrow a \rightarrow fail \rightarrow STOP$$
$$T_T(\langle\,\checkmark\,\rangle, a) = pass \rightarrow a \rightarrow fail \rightarrow STOP$$
$$T_T(\langle\, b\,\rangle \frown s, a) = inc \rightarrow b \rightarrow T_T(s, a)$$

Execution $Execution^P_Q(T)$ of a test for Q, against a specification P, is described by the CSP process $(Q \,[\![\, \alpha P \,]\!]\, T)\backslash\alpha P$. The exhaustive test set $Exhaust_T(P)$ for trace refinement of P contains all $T_T(s, a)$ formed from a trace $s \in traces(P)$, and forbidden continuation a. Proof of exhaustivity is in [3].

4.2 Local distributed testing

For simplicity we identify users by numbers, and index their events by these numbers. Since users need not have disjoint alphabets, an event may be indexed by several numbers. Moreover, we augment the set of events U of every user, with the set $V_U = \{pass_U, fail_U, inc_U\}$ of events for local verdicts.

Given $T_T(s, a)$ and a user U, we derive a local test $T_T(s, a)|_U$ by removing from $T_T(s, a)$ all events unobservable by U and associated verdicts.

Example 15. We consider users 1 and 2 defined by $\{a_1, b_1\}$ and $\{a_2, b_2\}$, and a specification $P = a_1 \rightarrow a_2 \rightarrow b_1 \rightarrow b_2 \rightarrow STOP$. We have a global test $T_T(\langle a_1, a_2\rangle, a_1) = inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow pass \rightarrow a_1 \rightarrow fail \rightarrow STOP$. The local tests are $T_T(\langle a_1, a_2\rangle, a_1)|_1 = inc_1 \rightarrow a_1 \rightarrow inc_1 \rightarrow a_1 \rightarrow fail_1 \rightarrow STOP$ and $T_T(\langle a_1, a_2\rangle, a_1)|_2 = inc_2 \rightarrow a_2 \rightarrow pass_2 \rightarrow STOP$, in this case. □

For every traces-refinement test T, that is, a CSP process in the range of the function T_T, and a user U, we define $T|_U$ inductively as follows.

$$(inc \rightarrow T)|_U = inc_U \rightarrow T|_U$$
$$(a \rightarrow v \rightarrow T)|_U = a \rightarrow v_U \rightarrow T|_U, \text{ if } a \in U \setminus V_U, v \in V$$
$$(a \rightarrow v \rightarrow T)|_U = T|_U, \text{ if } a \notin U, v \in V$$
$$STOP|_U = STOP$$

The global tests for the empty trace start already with a *pass* event. The corresponding local tests are defined as follows.

$$(pass \rightarrow a \rightarrow fail \rightarrow STOP)|_U = pass_U \rightarrow a \rightarrow fail_U \rightarrow STOP, \text{ if } a \in U$$
$$(pass \rightarrow a \rightarrow fail \rightarrow STOP)|_U = inc_U \rightarrow STOP, \text{ if } a \notin U$$

The distributed execution $Execution^P_Q(T, \mathcal{A})$ of local tests corresponding to a global test T, with set of users $\mathcal{A}$, for implementation Q, against a specification P, is described by the CSP process $(Q \,[\![\, \alpha P \,]\!]\, (\,\|\, U : \mathcal{A} \bullet [U]\, T|_U))\backslash\alpha P$. The test $(\,\|\, U : \mathcal{A} \bullet [U]\, T|_U)$ runs the local tests $T|_U$ for users U in $\mathcal{A}$ in parallel, with synchronisation only on common (original) events. Since the verdict events V_U of each user are different, each test produces its own verdict. The overall verdict arising from the experiment is *failure* if any user U observes a $fail_U$. If not, it is a *success* if any user observes a $pass_U$, and *inconclusive* otherwise.

We need to observe, however, that the local tests are not necessarily sound.

Example 16. We consider again users $\{a_1, b_1\}$ and $\{a_2, b_2\}$. For the specification $P = a_1 \rightarrow a_2 \rightarrow STOP \ \Box \ a_2 \rightarrow STOP$, we have a trace $\langle a_2 \rangle$, with forbidden continuation a_1. We have a global test $inc \rightarrow a_2 \rightarrow pass \rightarrow a_1 \rightarrow fail \rightarrow STOP$. The local tests are $inc_1 \rightarrow a_1 \rightarrow fail_1 \rightarrow STOP$ and $inc_2 \rightarrow a_2 \rightarrow pass_2 \rightarrow STOP$. If the system performs the trace $\langle a_1, a_2 \rangle$, the verdict of these tests is *failure*, even though $\langle a_1, a_2 \rangle$ is a trace of the specification P. $\Box$

Here we have a controllability problem: the second local test should not start until after event a_2. Under certain conditions, soundness is guaranteed: for instance, if for every component $a \rightarrow v \rightarrow b \rightarrow T$ of the test, where a and b are not verdict events, but v is, at least one user can observe both a and b. In the next section we explore the use of coordination messages to address this issue.

4.3 Coordination messages and traces refinement

The approach presented here is inspired by that in Hierons [5]. First, we introduce coordination messages as events $coord.i.j$ observable by users i and j. The role of such an event is to allow the tester i to warn the tester j that the event a_i has just been performed and j is entitled to propose the event a_j.

For a global test $T_T(s, a)$, defined from a trace s and a forbidden event a observable by a user k, coordination messages are inserted in the local tests as follows. For every pair a_i and a_j of consecutive events of s observed by different users i and j, the event $coord.i.j$ is inserted in $T_T(s, a)|_i$ after a_i and in $T_T(s, a)|_j$ before a_j. If the user i that observes the last event a_i of s is not k, then $coord.i.k$ is inserted in $T_T(s, a)|_i$ after a_i and in $T_T(s, a)|_k$ before a.

Example 17. We consider the global test $T_T(\langle a_1, a_2 \rangle, a_1)$ from Example 15. We get $inc_1 \rightarrow a_1 \rightarrow coord.1.2 \rightarrow inc_1 \rightarrow coord.2.1 \rightarrow a_1 \rightarrow fail_1 \rightarrow STOP$ as the coordinated version of the local test $T_T(\langle a_1, a_2 \rangle, a_1)|_1$. That of $T_T(\langle a_1, a_2 \rangle, a_1)|_2$ is $inc_2 \rightarrow coord.1.2 \rightarrow a_2 \rightarrow coord.2.1 \rightarrow pass_2 \rightarrow STOP$. $\Box$

The function $C^i(T)$ that defines the annotated local test for user i from a global test T can be defined inductively as follows.

$$
\begin{array}{l}
C^i(inc \rightarrow T) = inc_i \rightarrow C^i(T) \\
C^i(a_i \rightarrow v \rightarrow b_i \rightarrow T) = a_i \rightarrow v_i \rightarrow C^i(b_i \rightarrow T) \\
C^i(a_i \rightarrow v \rightarrow a_k \rightarrow T) = a_i \rightarrow coord.i.k \rightarrow v_i \rightarrow C^i(a_k \rightarrow T), \text{ if } k \neq i \\
C^i(a_j \rightarrow v \rightarrow a_i \rightarrow T) = coord.j.i \rightarrow C^i(a_i \rightarrow T), \text{ if } j \neq i \\
C^i(a_j \rightarrow v \rightarrow a_k \rightarrow T) = C^i(a_k \rightarrow T), \text{ if } j \neq i, k \neq i \\
C^i(a_i \rightarrow fail \rightarrow STOP) = a_i \rightarrow fail_i \rightarrow STOP \\
C^i(a_j \rightarrow fail \rightarrow STOP) = STOP, \text{ if } j \neq i \\
C^i(pass \rightarrow a_i \rightarrow fail \rightarrow STOP) = pass_i \rightarrow a_i \rightarrow fail_i \rightarrow STOP \\
C^i(pass \rightarrow a_j \rightarrow fail \rightarrow STOP) = inc_i \rightarrow STOP, \text{ if } j \neq i
\end{array}
$$

The distributed test is defined by $(\ \|\ U : \mathcal{A} \bullet [AC(U)]C^U(T)) \setminus \{\!|coord|\!\}$. As before, we have a parallel composition of the local tests. $AC(U)$ is the alphabet of $C^U(T)$, including U, and the events $coord.U.i$ and $coord.i.U$, for every user

i in $\mathcal{A}$. The set $\{\!|coord|\!\}$ of all coordination events is hidden, as they are used for interaction among the tests, but not with the system under test.

The role of the coordination messages as defined above is to preserve the global order of events when using local tests. Since the synchronisation constraints they introduce force the local tests to follow the order of the global tests, which have been defined to exhaustively test for traces refinement, they introduce a distributed coordinated way to test for such refinement. It is at the price of their proliferation, which seems unavoidable in general, if traces refinement is required and if there is no way of performing global testing. It is likely, however, that there are some types of systems where the coordination of distributed tests is less costly, for instance when global traces have some locality properties, like long local subtraces, and few switches between users.

5 Conclusions

This paper has explored distributed testing from a CSP specification. While there has been much work on distributed testing from an IOTS or a finite state machine, the use of CSP introduces new challenges. For example, since some refinement relations for CSP assume that we can observe failures in addition to traces, there is a need to incorporate failures into the framework. The distinction between internal choice and external choice also introduces interesting issues.

We have considered two situations. In the first, the testers are distributed, but their observations can be brought together. This leads to the notion of cooperating refinement. In this case, it is necessary to decide when the observations can be brought together, since the testers need to know that they are reporting observations regarding the same trace. We have, therefore, defined the notion of a complete run, which is either infinite or a trace that terminates. Since the failure sets are the same after all complete runs, there is no value in observing them, and so we only observe projections of traces.

In the alternative situation, the testers act entirely independently. Here it is sufficient for the observations made by each tester to be consistent with the specification, even if the set of observations is not. A single tester can observe traces and failures, and as a result we have defined independent traces refinement, under which only traces are observed, and independent failures refinement, under which traces and failures are observed. We have also considered test generation and showed how coordination messages might be used to make tests sound.

There are several avenues for future work. First, under cooperating refinement the testers do not observe failures and it would be interesting to find ways of incorporating information regarding failures. In addition, CSP does not distinguish between inputs and outputs. It is, however, possible to include such a distinction through the notion of non-delayable events to model outputs. This is certainly a crucial step to enable the study of more elaborate concerns regarding observability and control. Recent work has shown how the notion of a scenario, which is a sequence of events after which the testers might synchronise, can be used in distributed testing from an IOTS and it should be possible to introduce

scenarios into CSP. Additionally, the work in [3] considers *conf* as well as traces refinement; distributed testing for *conf* might raise interesting issues for coordination. Finally, tools for test generation and case studies are needed to explore the applications of the theory presented here.

Acknowledgments Cavalcanti and Gaudel are grateful for the support of the Royal Academy of Engineering. We acknowledge invaluable contributions by Manuel Núñez to a previous version of this paper and to many discussions.

References

1. E. Brinksma, L. Heerink, and J. Tretmans. Factorized test generation for multi-input/output transition systems. In *11th IFIP Workshop on Testing of Communicating Systems*, pages 67 – 82. Kluwer Academic Publishers, 1998.
2. L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *IST*, 41(11 – 12):767 – 780, 1999.
3. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *ICFEM*, volume 4789 of *LNCS*, pages 151 – 170. Springer-Verlag, 2007.
4. R. Dssouli and G. v. Bochmann. Error detection with multiple observers. In *PSTV*, pages 483 – 494. Elsevier Science, 1985.
5. R. M. Hierons. Overcoming controllability problems in distributed testing from an input output transition system. Submitted. Available at `people.brunel.ac.uk/ csstrmh/coord.pdf`
6. R. M. Hierons, M. G. Merayo, and M. Núñez. Controllable test cases for the distributed test architecture. In *6th ATVA*, volume 5311 of *LNCS*, pages 201–215. Springer, 2008.
7. R. M. Hierons, M. G. Merayo, and M. Nunez. Implementation relations for the distributed test architecture. In *20th TESTCOM/FATES*, volume 5047 of *LNCS*, pages 200 – 215. Springer, 2008.
8. R. M. Hierons, M. G. Merayo, and M. Núñez. Scenarios-based testing of systems with distributed ports. *Software - Practice and Experience*, 2011. Accepted for publication, doi: 10.1002/spe.1062.
9. R. M. Hierons and M. Nunez. Scenarios-based testing of systems with distributed ports. In *10th QSIC*, 2010.
10. J. Jacob. Refinement of shared systems. In *The Theory and Practice of Refinement*, pages 27 – 36. Butterworths, 1989.
11. C. Jard. Synthesis of distributed testers from true-concurrency models of reactive systems. *IST*, 45(12):805– 814, 2003.
12. C. Jard, T. Jéron, H. Kahlouche, and C. Viho. Towards automatic distribution of testers for distributed conformance testing. In *FORTE*, pages 353–368. Kluwer Academic Publishers, 1998.
13. S. Pickin, C. Jard, Y. L. Traon, T. Jéron, J.-M. Jézéquel, and A. L. Guennec. System test synthesis from UML models of distributed software. In *Formal Techniques for Networked and Distributed Systems*, volume 2529 of *LNCS*, pages 97 – 113. Springer, 2002.
14. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
15. B. Sarikaya and G. v. Bochmann. Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications*, 32:389 – 395, 1984.
16. H. Ural and C. Williams. Constructing checking sequences for distributed testing. *FAC*, 18(1):84 – 101, 2006.