Logical Methods in Computer Science
Volume 19, Issue 1, 2023, pp. 1:1–1:31
https://lmcs.episciences.org/

Submitted Dec. 13, 2016
Published Jan. 05, 2023

MULTIPARTY TESTING PREORDERS

ROCCO DE NICOLA [a] AND HERNÁN MELGRATTI [b]

[a] IMT School for Advanced Studies Lucca, Italy
e-mail address: rocco.denicola@imtlucca.it

[b] DC, FCEyN, Universidad de Buenos Aires - CONICET, Argentina
e-mail address: hmelgra@dc.uba.ar

Abstract. Variants of the must testing approach have been successfully applied in service oriented computing for analysing the compliance between (contracts exposed by) clients and servers or, more generally, between two peers. It has however been argued that multiparty scenarios call for more permissive notions of compliance because partners usually do not have full coordination capabilities. We propose two new testing preorders, which are obtained by restricting the set of potential observers. For the first preorder, called uncoordinated, we allow only sets of parallel observers that use different parts of the interface of a given service and have no possibility of intercommunication. For the second preorder, that we call individualistic, we instead rely on parallel observers that perceive as silent all the actions that are not in the interface of interest. We have that the uncoordinated preorder is coarser than the classical must testing preorder and finer than the individualistic one. We also provide a characterisation in terms of decorated traces for both preorders: the uncoordinated preorder is defined in terms of must-sets and Mazurkiewicz traces while the individualistic one is described in terms of classes of filtered traces that only contain designated visible actions and must-sets.

1. Introduction

A desired property of communication-centered systems is the graceful termination of the processes involved in a multiparty interaction, i.e., every possible interaction ends successfully, in the sense that there are neither messages waiting forever to be sent nor sent messages which are never received. The theories of session types [THK94, HVK98] and of contracts [CGP08, CGP09, BZ09, LP07] are commonly used to ensure such kind of properties. The key idea behind both approaches is to associate each process a with type (or *contract*) that gives an abstract description of its external, visible behaviour and to use type checking to verify the correctness of behaviours.

Processes are often defined as sequential nondeterministic CCS processes [Mil89] describing the offered communications, and are built-up from send and receive actions. These

2012 ACM CCS: [**Theory of computation**]: Models of computation—Concurrency—Process Calculi.
Key words and phrases: Must-testing preorder; Multiparty composition; Mazurkiewicz traces.

* This is an extended and revised version of a paper with same title that appeared in the proceedings of the 10th International Symposium on Trustworthy Global Computing, Vol. 9533 of LNCS, Springer 2016.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-19(1:1)2023

activities are abstractly represented either as output and input actions that take place over a set of channels or as internal τ actions. Basic actions can be composed sequentially (prefix operator ".") or as alternatives (non deterministic choice "+"). Typically, the language for describing processes does not have any operator for parallel composition. It is assumed that all possible interleavings are made explicit in the description of the service and communication is only used for modelling the interaction among different processes.

In *client-server* scenarios, i.e., in settings involving just two processes, variants of the must testing preorder has been used to compare alternative implementations of servers and clients [BH13]. Technically, two given processes p and q are related via the must preorder ($p \sqsubseteq_{\mathsf{must}} q$) if q satisfies all observers that are satisfied by p. Consequently, p and q are considered equivalent ($p \approx_{\mathsf{must}} q$) if they satisfy exactly the same observers. Standardly, an observer is a unique (sequential) process that runs in parallel with the tested process and, consequently, all interactions with the tested process are handled by a unique, central process, i.e., the observer itself.

If one considers a multiparty setting, each process may concurrently interact with several other partners and its interface is often (logically) partitioned by allowing each partner to communicate only through a dedicated part of the interface. Consider the following scenario involving three partners: an organisation (the broker) that sells goods produced by a different company (the producer) to a specific customer (the client). The behaviour of the broker can be described by the following process:

$$B = req.\overline{order}.\overline{inv}.\mathbf{0}$$

The broker accepts requests on channel *req* and then places an order to the producer with the message $\overline{order}$ and sends an invoice to the client with the message $\overline{inv}$. A client may behave as the process C below, which first sends a request on channel *req* and then expects the invoice on channel *inv*, i.e.,

$$C = \overline{req}.inv.\mathbf{0}$$

A producer may be modelled by a process that simply accepts an order over channel *ord*, i.e.,

$$P = order.\mathbf{0}$$

In this scenario, the broker uses the channels *req* and *inv* to interact with the client, while it interacts with the producer over the channel *order*. Moreover, the client and the producer do not know each other and are completely independent. Hence, the order in which messages $\overline{order}$ and $\overline{inv}$ are sent is completely irrelevant for them. In fact, they would be equally happy with a broker defined as follows:

$$B' = req.\overline{inv}.\overline{order}.\mathbf{0}$$

Nevertheless, these two different implementations are not considered must-equivalent. In these situations, the classical must testing preorder turns out to be unnecessarily discriminating.

The main goal of this paper is to introduce alternative, less discriminating, preorders that take into account the distributed nature of the observers and their possibly limited coordination and interaction capabilities. A first preorder, called *uncoordinated must preorder*, is obtained by assuming that a set of observers of a given process interact with it via fully disjoint sets of ports, i.e., they use different parts of its interface, have no possibility of intercommunication, and all of them terminate successfully in every possible interaction. It is however worth noting that these assumptions about the absence of communication

among observers do not fully eliminate the possibility for observers of being mutually influenced, e.g., when one observer does not enable a communication on some ports. Due to this, it is possible to differentiate B from B' above when one of the observers refuses to synchronise over a port, e.g., if the client does not enable the synchronisation over the channel *inv*. Consider a client $C' = \overline{req}.\mathbf{0}$ that sends a request and terminates without accepting the invoice. While P and C' always terminate their interaction with B, this is not the case when interacting with B' because communication over channel *order* is never enabled. Consequently, these two implementations of the broker are distinguished by the uncoordinated must preorder. However, the uncoordinated must preorder allows for the reordering of actions. For instance, the following two implementations of the broker are considered equivalent under the uncoordinated must preorder.

$$B'' = req.(\overline{order}.\overline{inv}.\mathbf{0} + \overline{order}.\mathbf{0} + \overline{inv}.\mathbf{0})$$
$$B''' = req.(\overline{inv}.\overline{order}.\mathbf{0} + \overline{order}.\mathbf{0} + \overline{inv}.\mathbf{0})$$

We remark that the processes C' and P are not able to distinguish B'' from B''', because they both terminate when interacting with either B'' and B'''. We also note that a client behaving as described by C will not be satisfied neither by B'' nor B''' because they both may decide not to communicate over *inv*.

The second preorder, which we call *individualistic must preorder*, allows observers to take for granted the execution of those actions of the process that are not explicitly of interest for them (i.e., not in their alphabet). For instance, a client in the previous scenario assumes that the producer will always enable the communication over the channel *order*. In the individualistic must preorder, the processes B and B' turn out to be indistinguishable.

The preorders are, as usual, defined in terms of the outcomes of experiments by specific sets of observers. For defining the uncoordinated must preorder, we allow only sets of parallel observers that cannot intercommunicate and do challenge processes via disjoint parts of their interface. For defining the individualistic must preorder, we instead rely on parallel observers that, again, cannot intercommunicate but in addition perceive as silent all the actions that are not part of the interface of their interest. This is instrumental to avoid that a specific observer recovers information about other involved observers. As expected, we have that the uncoordinated preorder is coarser than the classical must testing preorder and finer than the individualistic one.

Just like for classical testing preorders, we provide a characterisation for both new preorders in terms of decorated traces, which avoids dealing with universal quantifications over the set of observers whenever a specific relation between two processes has to be established. The alternative characterisations make it even more evident that our preorders permit action reordering. Indeed, the uncoordinated preorder is defined in terms of *Mazurkiewicz traces* [Maz95] while the individualistic one is described in terms of classes of traces quotiented via specific sets of visible actions. We would like to remark that our preorders are different from those defined in [BZ08, Pad10, MYH09], which also permit action reordering by relying on buffered communication; additional details will be provided in Section 7.

Synopsis The remainder of this paper is organised as follows. In Section 2 we recall the basics of the classical must testing approach. In Section 3 and Section 4 we present the theory of uncoordinated and individualistic must testing preorders and their characterisation in terms of traces. In Section 5 we show that the uncoordinated preorder is coarser than the must testing preorder but finer than the individualistic one. In Section 6 we describe a Prolog implementation of the uncoordinated and individualistic preorders for the finite fragment of

our specification language and use it for analysing a scenario involving a replicated data store. Finally, we discuss some related work and future developments in Section 7.

This paper is a revised and extended version of [NM15]. We fix an incorrect characterisation of the uncoordinated preorder [NM15], and provide full proofs of previously published results. In addition, we give a prototype implementation in Prolog of the alternative characterisation of the proposed preorders for the fragment of the calculus with only finite processes and illustrate the usability of the proposed preorders by using them to reason on different implementations of components in a replicated data store (Section 6).

2. Processes and testing preorders

Let $\mathcal{N}$ be a countable set of action names, ranged over by $a, b, \ldots$. As usual, we write co-names in $\overline{\mathcal{N}}$ as $\overline{a}, \overline{b}, \ldots$ and assume $\overline{\overline{a}} = a$. We will use α, β to range over $\mathsf{Act} = (\mathcal{N} \cup \overline{\mathcal{N}})$. Moreover, we consider a distinguished internal action τ not in Act and use μ to range over $\mathsf{Act} \cup \{\tau\}$. We fix the language for defining processes as the sequential fragment of CCS extended with a *success* operator, as specified by the following grammar.

$$p, q ::= \mathbf{0} \mid \mathbf{1} \mid \mu.p \mid p + q \mid X \mid \mathbf{rec}_X.p$$

The process $\mathbf{0}$ stands for the terminated process, $\mathbf{1}$ for the process that reports success and then terminates, and $\mu.p$ for a process that executes μ and then continues as p. Alternative behaviours are specified by terms of the form $p + q$, while recursive ones are introduced by terms like $\mathbf{rec}_X.p$. We denote by $\mathcal{P}$ the set of all processes. We write $\mathtt{n}(p)$ for the set of names $a \in \mathcal{N}$ such that either a or $\overline{a}$ occur in p.

The operational semantics of processes is given in terms of a labelled transition system (LTS) $p \xrightarrow{\lambda} q$ with $\lambda \in \mathsf{Act} \cup \{\tau, \checkmark\}$, where $\checkmark$ signals the successful termination of an execution.

Definition 2.1 (Transition relation)**.** The transition relation on processes, noted $\xrightarrow{\lambda}$, is the least relation satisfying the following rules

$$\mathbf{1} \xrightarrow{\checkmark} \mathbf{0} \qquad \mu.p \xrightarrow{\mu} p \qquad \frac{p \xrightarrow{\lambda} p'}{p + q \xrightarrow{\lambda} p'} \qquad \frac{q \xrightarrow{\lambda} q'}{p + q \xrightarrow{\lambda} q'} \qquad \frac{p[\mathbf{rec}_X.p/X] \xrightarrow{\lambda} p'}{\mathbf{rec}_X.p \xrightarrow{\lambda} p'}$$

Multiparty applications, named *configurations*, are built by composing processes concurrently. Formally, configurations are given by the following grammar.

$$c, d, o ::= p \mid c \| d$$

We denote by $\mathcal{O}$ the set of all configurations. We sometimes write $\Pi_{i \in 0..n} p_i$ for the parallel composition $p_0 \parallel \ldots \parallel p_n$. The operational semantics of configurations, which accounts for the communication between processes, is obtained by extending the LTS in Definition 2.1 with the following rules:

$$\frac{c \xrightarrow{\mu} c'}{c \parallel d \xrightarrow{\mu} c' \parallel d} \qquad \frac{d \xrightarrow{\mu} d'}{c \parallel d \xrightarrow{\mu} c \parallel d'} \qquad \frac{c \xrightarrow{\alpha} c' \quad d \xrightarrow{\overline{\alpha}} d'}{c \parallel d \xrightarrow{\tau} c' \parallel d'} \qquad \frac{c \xrightarrow{\checkmark} c' \quad d \xrightarrow{\checkmark} d'}{c \parallel d \xrightarrow{\checkmark} c' \parallel d'}$$

All rules are standard apart for the last one that is not present in [NH84]. This rule states that the concurrent composition of processes can report success only when all processes in the composition do so.

We write $c \xrightarrow{\lambda}$ when there exists c' such that $c \xrightarrow{\lambda} c'$; $\Rightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$; $c \stackrel{\lambda}{\Longrightarrow} c'$ for $\lambda \in \mathsf{Act} \cup \{\checkmark\}$ and $c \Longrightarrow \xrightarrow{\lambda} \Longrightarrow$; $c \stackrel{\lambda_0 \ldots \lambda_n}{\Longrightarrow} c'$ for $c \stackrel{\lambda_0}{\Longrightarrow} \ldots \stackrel{\lambda_n}{\Longrightarrow} c'$,

and $c \stackrel{s}{\Longrightarrow}$ with $s \in (\mathsf{Act} \cup \{\checkmark\})^*$ if there exists c' such that $c \stackrel{s}{\Longrightarrow} c'$. We say that $c_o \stackrel{\mu_0}{\longrightarrow} \ldots c_i \stackrel{\mu_i}{\longrightarrow} \ldots$ is successful when there exists j s.t. $c_j \stackrel{\checkmark}{\longrightarrow}$; it is unsuccessful otherwise.

We write $\mathsf{str}(c)$ and $\mathsf{init}(c)$ to denote the sets of strings and enabled actions of c, defined as follows

$$\mathsf{str}(c) = \{s \in (\mathsf{Act} \cup \{\checkmark\})^* \mid c \stackrel{s}{\Longrightarrow}\} \qquad \mathsf{init}(c) = \{\lambda \in \mathsf{Act} \cup \{\checkmark\} \mid c \stackrel{\lambda}{\Longrightarrow}\}$$

As behavioural semantics, we consider the must-testing preorder [NH84], which is defined in terms of the computations of a process under test p and an observer o. A computation of $p \parallel o$ is a sequence of τ transitions, i.e.,

$$p \parallel o = p_0 \parallel o_0 \stackrel{\tau}{\rightarrow} \ldots \stackrel{\tau}{\rightarrow} p_k \parallel o_k \stackrel{\tau}{\rightarrow} \ldots$$

A computation is *maximal* if it is either infinite or its last term $p_n \parallel o_n$ is such that $p_n \parallel o_n \stackrel{\tau}{\not\rightarrow}$. We say it is *observer-successful* if there exists $j \geq 0$ such that $o_j \stackrel{\checkmark}{\longrightarrow}$, and *observer-unsuccessful* otherwise.

Definition 2.2 (must)**.** We write p must o iff for each maximal computation of $p \parallel o$ is observer-successful.

The notion of passing a test (or satisfying an observer) represents the fact that an observer built-up from the parallel composition of processes is able to report success in every possible interaction with the process under test. It is then natural to compare processes according to their capacity to satisfy observers.

The standard framework of [NH84] can be recovered by considering only observers without parallel composition.

Definition 2.3 (must preorder)**.** $p \sqsubseteq_{\mathsf{must}} q$ iff $\forall r \in \mathcal{P} : p$ must r implies q must r. We write $p \approx_{\mathsf{must}} q$ when both $p \sqsubseteq_{\mathsf{must}} q$ and $q \sqsubseteq_{\mathsf{must}} p$.

2.1. **Semantic characterisation.** The must testing preorder has been characterised in terms of (i) the sequences of actions that a process may perform, and (ii) the possible sets of actions that it may perform after executing a particular sequence of actions [NH84]. This characterisation relies on a few auxiliary predicates and functions that are presented below. A process p *diverges*, written $p \Uparrow$, when it exhibits an infinite, internal computation $p \stackrel{\tau}{\rightarrow} p_0 \stackrel{\tau}{\rightarrow} p_1 \stackrel{\tau}{\rightarrow} \ldots$. We say p *converges*, written $p \Downarrow$, otherwise. For $s \in \mathsf{Act}^*$, the convergence predicate is inductively defined by the following rules:

- $p \Downarrow \epsilon$ if $p \Downarrow$.
- $p \Downarrow \alpha.s$ if $p \Downarrow$ and $p \stackrel{\alpha}{\Longrightarrow} p'$ implies $p' \Downarrow s$.

The *residuals* of a process p (or a set of processes P) after the execution of $s \in \mathsf{Act}^*$ is given by the following equations

- $(p \text{ after } s) = \{p' \mid p \stackrel{s}{\Longrightarrow} p'\}$.
- $(P \text{ after } s) = \{p' \mid p \in P,\ p' \in (p \text{ after } s)\}$.

Definition 2.4 (Must-set)**.** A *must-set* of a process p (or set of processes P) is $L \subseteq \mathsf{Act}$, and L finite such that

- p MUST L iff $\forall p'$ such that $p \Longrightarrow p'$, $\exists \alpha \in L$ such that $p' \stackrel{\alpha}{\Longrightarrow}$.
- P MUST L iff $\forall p \in P.p$ MUST L.

Then, the must testing preorder can be characterised in terms of strings and must-sets as follows.

Definition 2.5. $p \preceq_{\mathsf{must}} q$ if for every $s \in \mathsf{Act}^*$, for all finite $L \subseteq \mathsf{Act}$, if $p \Downarrow s$ then
- $q \Downarrow s$.
- (p after s) MUST L implies (q after s) MUST L.

Theorem 2.6 [NH84, Theorem 6.4.5]**.** $\sqsubseteq_{\mathsf{must}} = \preceq_{\mathsf{must}}$.

3. A testing preorder with uncoordinated observers

The must testing preorder is defined in terms of the tests that each process is able to pass. Remarkably, the classical setting can be formulated by considering only sequential tests (see the characterisation of minimal tests in [NH84]). Each sequential test is a unique, centralised process that handles all the interaction with the process under test and, therefore, has a complete view of the externally observable behaviour of the process. For this reason, we refer to the classical must testing preorder as a *centralised preorder*.

Multiparty interactions are generally structured in such a way that pairs of partners communicate through dedicated channels, for example, when relying on partner links in service oriented models or buffers in communicating machines [BBO12]. Conceptually, the interface (i.e., the set of channels) of a process is partitioned and the process interacts with each partner by using only specific sets of channels in its interface. In addition, there are scenarios (as the one discussed in Section 1) in which partners frequently do not know each other and cannot communicate directly. As a direct consequence, the partners of a process cannot establish causal dependencies among actions that take place over different parts of the interface. These constraints reduce the discriminating power of partners and call for equivalences that equate processes that cannot be distinguished by sets of independent sequential processes.

Example 3.1. Consider the classical scenario for planning a trip. A user U interacts with a broker B, which is responsible for booking flights provided by a service F and hotel rooms available at service H. The expected interaction can be described as follows: U makes a booking request by sending a message req to B (we will just describe the interaction and abstract away from data details such as trip destination, departure dates and duration). Depending on the request, B may internally decide to contact service F (for booking just a flight ticket), H (for booking rooms) or both. Service B uses channels $reqF$ and $reqH$ to respectively contact F and H (for the sake of simplicity, we assume that any request to F and H will be granted). Then, the expected behaviour of B can be described with the following process. (As usual, τ actions and $+$ are combined to model internal, non-deterministic choices in a process.)

$$B_0 \stackrel{def}{=} req.(\tau.\overline{reqF}.\mathbf{0} + \tau.\overline{reqH}.\mathbf{0} + \tau.\overline{reqH}.\overline{reqF}.\mathbf{0})$$

In this process, the third branch represents B's choice to contact first H and then F. Nevertheless, the other partners (U, F and H) are not affected in any way by this choice, thus they would be equally happy with alternative definitions such as:

$$B_1 \stackrel{def}{=} req.(\tau.\overline{reqF}.\mathbf{0} + \tau.\overline{reqH.\mathbf{0} + \tau.\overline{reqF}.\overline{reqH}.\mathbf{0}})$$
$$B_2 \stackrel{def}{=} req.(\tau.\overline{reqF}.\mathbf{0} + \tau.\overline{reqH}.\mathbf{0} + \tau.\overline{reqH}.\overline{reqF}.\mathbf{0} + \tau.\overline{reqF}.\overline{reqH}.\mathbf{0})$$

B_0, B_1 and B_2 are distinguished by the must testing equivalence. It suffices to consider $o_0 = \overline{req}.(\tau.\mathbf{1} + reqF.(\tau.\mathbf{1} + reqH.\mathbf{0}))$ for showing that $B_0 \not\sqsubseteq_{\mathsf{must}} B_1$ and that $B_0 \not\sqsubseteq_{\mathsf{must}} B_2$, and use $o_1 = \overline{req}.(\tau.\mathbf{1} + reqH.(\tau.\mathbf{1} + reqF.\mathbf{0}))$ for proving that $B_1 \not\sqsubseteq_{\mathsf{must}} B_2$. □

This section is devoted to the definition and characterisation of a preorder, called *uncoordinated must preorder*, that is coarser than the classical must preorder and relates processes that cannot be distinguished by distributed contexts. The uncoordinated must preorder is obtained by restricting the set of observers to parallel processes that do not share channels. We will say $\mathbb{I} = \{I_i\}_{i\in 0\ldots n}$ is an interface whenever $\mathbb{I}$ is a partition of Act and $\forall \alpha \in \mathsf{Act}$, $\alpha \in I_i$ implies $\overline{\alpha} \in I_i$. In the rest of this paper we will usually write only the relevant part of an interface. For instance, we will write $\{\{a\},\{b\}\}$ for any interface $\{I_0, I_1\}$ such that $a \in I_0$ and $b \in I_1$. Then, the observers used by the uncoordinated must testing preorder are introduced by the following definition.

Definition 3.2 (Uncoordinated observer)**.** Let $\mathbb{I} = \{I_i\}_{i\in 0\ldots n}$ be an interface. A process $\Pi_{i\in 0..n} o_i = o_0 \parallel \ldots \parallel o_n$ is an *uncoordinated observer over* $\mathbb{I}$ if $\mathtt{n}(o_i) \subseteq I_i$ for all $i \in 0\ldots n$.

We say o is an uncoordinated observer and omit the interface when no confusion arises. In our setting, which does not involve name mobility, the fact that $\mathbb{I} = \{I_i\}$ is a partition of Act and $\mathtt{n}(o_i) \subseteq I_i$ suffices to avoid a direct communication among the processes of an uncoordinated observer. As a consequence, a distributed observer cannot impose a total order between actions that are controlled by different processes of the observer. Indeed, the executions of a distributed observer are the interleavings of the executions of all processes $\{o_i\}_{i\in 0..n}$ (this property is formally stated in Section 3.1, Lemma 3.6). We remark that a configuration does report success (i.e., perform action ✓) only when all processes in the composition do report success; consequently an uncoordinated observer reports success when all its components report success simultaneously. Our definition of success deviates from the original setup of [NH84]. If success were not synchronised, e.g., every process would pass the observer $o = a.\mathbf{0} \parallel \mathbf{1}$ because o would be able to report success immediately. This is not the case in our setting. In fact, each component of an uncoordinated observer accounts for the view that a particular partner has about the process under test, and we expect every component of the observer to be able to report success when a process passes a test.

Definition 3.3 (Uncoordinated must preorder $\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}}$)**.** Let $\mathbb{I} = \{I_i\}_{i\in 0\ldots n}$ be an interface. We say $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$ iff for all uncoordinated observer o over $\mathbb{I}$, p must o implies q must o. We write $p \approx^{\mathbb{I}}_{\mathsf{unc}} q$ when both $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$ and $q \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} p$.

Example 3.4. Consider the scenario presented in Example 3.1 and the following interface $\mathbb{I} = \{\{req\},\{reqF\},\{reqH\}\}$ for the process B that thus interacts with each of the other partners by using a dedicated part of its interface. It can be shown that the three definitions for B in Example 3.1 are equivalent when considering the uncoordinated must testing preorder, i.e., $B_0 \approx^{\mathbb{I}}_{\mathsf{unc}} B_1 \approx^{\mathbb{I}}_{\mathsf{unc}} B_2$. The actual proof, which uses the (trace-based) alternative characterization of the preorder, is deferred to Example 3.14. □

3.1. **Semantic characterisation.** We now characterise the uncoordinated must testing preorder in terms of traces and must-sets. In order to do that, we shift from strings to Mazurkiewicz traces [Maz86]. A *Mazurkiewicz trace* is a set of strings, obtained by permuting independent symbols. Traces represent concurrent computations, in which commuting symbols stand for actions that execute independently of one another and non-commuting symbols are causally dependent actions. We start by summarising the basics of the theory of traces in [Maz86].

Let $D \subseteq \mathsf{Act} \times \mathsf{Act}$ be a finite equivalence relation, called the *dependency relation*, that relates the actions that cannot be commuted. Thus, if $(\alpha, \beta) \in D$ then α and β

are considered causally dependent. Symmetrically, $I_D = (\mathsf{Act} \times \mathsf{Act}) \setminus D$ stands for the *independency* relation with $(\alpha, \beta) \in I_D$ meaning that α and β are concurrent.

The trace equivalence induced by the dependency relation D is the least congruence $\equiv_D$ on Act^* such that for all $\alpha, \beta \in \mathsf{Act} : (\alpha, \beta) \in I_D \implies \alpha\beta \equiv_D \beta\alpha$.

The equivalence classes of $\equiv_D$, denoted by $[s]_D$, are the Mazurkiewicz *traces*, namely the strings quotiented via $\equiv_D$. We remark that no action can commute with $\checkmark$ because I_D is defined over $\mathsf{Act} \times \mathsf{Act}$.

Let $\mathbb{I}$ be an interface, the *dependency relation induced by* $\mathbb{I}$ is $D = \bigcup_{I \in \mathbb{I}} I \times I$.

Example 3.5. Consider the interface $\mathbb{I} = \{\{req\}, \{reqF\}, \{reqH\}\}$ in Example 3.4. We recall that $\mathbb{I} = \{\{req\}, \{reqF\}, \{reqH\}\}$ is a convenient notation for any partition of Act such that $\forall \alpha \in \mathsf{Act}$, $\alpha \in I_i$ implies $\overline{\alpha} \in I_i$. Consequently,

$$\mathbb{I} = \{\{req, \overline{req}, \ldots\}, \{reqF, \overline{reqF}, \ldots\}, \{reqH, \overline{reqH}, \ldots\}\}$$

The dependency relation induced by $\mathbb{I}$ is

$$D = \{(req, req), (req, \overline{req}), (\overline{req}, req), (\overline{req}, \overline{req}), \ldots, (reqF, reqF), \ldots, (reqH, reqH), \ldots\}$$

Then, $(\alpha, \beta) \in I_D$ iff $(\alpha, \beta) \notin D$. The relation I_D basically states that actions that take place over channels that belong to different parts of the interface can commute. For instance, *req*, *reqH* and *reqF* are independent, and hence

$$req\ reqH\ reqF \equiv_D req\ reqF\ Reqh \equiv_D reqH\ reqF\ req \equiv_D \ldots$$

Consequently,

$$[req\ reqH\ reqF]_D = \{req\ reqH\ reqF,\ \ req\ reqF\ reqH,\ \ reqH\ reqF\ req,\ \ldots\}$$

On the contrary, actions that take place over channels in the same part of the interface are dependent and cannot commute. Hence, $req\ \overline{req} \not\equiv_D \overline{req}\ req$. □

Now we are able to characterise the behaviour of an uncoordinated observer. We start by formally stating that an uncoordinated observer reaches the same configuration after executing any of the strings in the same equivalence class. This result is instrumental to the proof of the alternative characterisation of the uncoordinated preorder.

Lemma 3.6. *Let $o = \Pi_{i \in 0..n} o_i$ be an uncoordinated observer over $\mathbb{I} = \{I_i\}_{i \in 0..n}$ and D the dependency relation induced by $\mathbb{I}$. Then, for all $s \in \mathsf{Act}^*$ and $t \in [s]_D$ we have $o \stackrel{s}{\Longrightarrow} o'$ iff $o \stackrel{t}{\Longrightarrow} o'$.*

Proof. The proof follows by induction on the length $|s| = |t| = n$.

- $\mathbf{n = 0, 1}$. Immediate, because $s = t$.
- $\mathbf{n > 1}$. By the Levi's Lemma for traces [Maz95, Theorem 1], any possible choice of v, w, x, y such that $s = vw$ and $t = xy$, implies that $v \equiv_D z_1 z_2$, $w \equiv_D z_3 z_4$, $x \equiv_D z_1 z_3$, $y \equiv_D z_2 z_4$ with $(z_2, z_3) \in I_D$. Consider a decomposition such that $|v|, |w|, |x|, |y| > 0$ (this is always possible, because $n > 1$). By inductive hypothesis on the reductions $o \stackrel{v}{\Longrightarrow} o''$ and $o'' \stackrel{w}{\Longrightarrow} o'$, we have $o \stackrel{v}{\Longrightarrow} o'' \stackrel{w}{\Longrightarrow} o'$ iff $o \stackrel{z_1}{\Longrightarrow} o_1 \stackrel{z_2}{\Longrightarrow} o_2 \stackrel{z_3}{\Longrightarrow} o_3 \stackrel{z_4}{\Longrightarrow} o'$. Since z_2 and z_3 are independent they take part on different components of the observer and $o_1 \stackrel{z_2}{\Longrightarrow} o_2 \stackrel{z_3}{\Longrightarrow} o_3$ iff $o_1 \stackrel{z_3}{\Longrightarrow} o_2' \stackrel{z_2}{\Longrightarrow} o_3$. Consequently, $o \stackrel{v}{\Longrightarrow} \stackrel{w}{\Longrightarrow} o'$ iff $o \stackrel{z_1}{\Longrightarrow} o_1 \stackrel{z_2}{\Longrightarrow} o_2 \stackrel{z_3}{\Longrightarrow} o_3 \stackrel{z_4}{\Longrightarrow} o'$ iff $o \stackrel{z_1}{\Longrightarrow} o_1 \stackrel{z_3}{\Longrightarrow} o_2' \stackrel{z_2}{\Longrightarrow} o_3 \stackrel{z_4}{\Longrightarrow} o'$. By inductive hypothesis on reductions $o \stackrel{z_1 z_3}{\Longrightarrow} o_2'$ and $o_2' \stackrel{z_2 z_4}{\Longrightarrow} o'$ we have $o \stackrel{z_1}{\Longrightarrow} o_1 \stackrel{z_2}{\Longrightarrow} o_2' \stackrel{z_3}{\Longrightarrow} o_3 \stackrel{z_4}{\Longrightarrow} o'$ iff $o \stackrel{x}{\Longrightarrow} \stackrel{y}{\Longrightarrow} o'$. □

Lemma 3.7. *Let $o = \Pi_{i\in 0..n} o_i$ be an uncoordinated observer over $\mathbb{I} = \{I_i\}_{i\in 0..n}$ and D the dependency relation induced by $\mathbb{I}$. Then, $\forall s \in \mathsf{Act}^*, t \in [s]_D$,*

(1) *$s \in \mathsf{str}(o)$ implies $t \in \mathsf{str}(o)$.*
(2) *$o \Downarrow s$ implies $o \Downarrow t$.*
(3) *$(o \text{ after } s)$ MUST L implies $(o \text{ after } t)$ MUST L.*
(4) *If there exists an unsuccessful computation $o \stackrel{s}{\Longrightarrow}$, then there exists an unsuccessful computation $o \stackrel{t}{\Longrightarrow}$.*

Proof. All items follow from Lemma 3.6. We illustrate (2). By contradiction. Assume $o \Downarrow s$ and $o \Uparrow t$. Then, there exist t_1, t_2 such that $t = t_1 t_2$, $o \stackrel{t_1}{\Longrightarrow} o'$ and $o' \Uparrow$. Without loss of generality, we assume that $o \stackrel{t_1}{\Longrightarrow} o'$ is minimal, i.e., $o = o_0 \xrightarrow{\mu_1} o_1 \ldots \xrightarrow{\mu_k} o_k = o'$ with $t_1 = \mu_1 \ldots \mu_k$ and $\forall i < k. o_i \Downarrow$. Since $o' = \Pi_{i\in 0..n} r_i$, there exists $h \in 0..n$ s.t. $r_h \Uparrow$. Then, take s_1 and s_2 such that $s = s_1 s_2$ and $s_1 \upharpoonright I_h = t_1 \upharpoonright I_h$. By the Levi's Lemma for traces [Maz95, Theorem 1], we have that $t_1 \equiv_D z_1 z_2$, $t_2 \equiv_D z_3 z_4$, $s_1 \equiv_D z_1 z_3$, $s_2 \equiv_D z_2 z_4$ with $(z_2, z_3) \in I_D$. Since $(z_2, z_3) \in I_D$, we have $o \stackrel{t_1}{\Longrightarrow} \stackrel{z_3}{\Longrightarrow} o''$ and $o'' \Uparrow$. By Lemma 3.6, $o \stackrel{s_1}{\Longrightarrow} \stackrel{z_2}{\Longrightarrow} o''$, which contradicts the hypothesis that $o \Downarrow s$. □

The alternative characterisation for the uncoordinated preorder follows along the lines of the one for the classical must testing preorder, when Mazurkiewicz traces are considered instead of strings of actions. For this reason, we extend the notions of transition relation, convergence and residuals to Mazurkiewicz traces.

We now focus on the definition of the transition relation, which is instrumental for the next definitions. We first note that, differently from centralised must testing preorder, string inclusion may not hold for the uncoordinated preorders. For instance, take $p = \tau.a.\mathbf{0} + \tau.b.\mathbf{0}$ and $q = a.b.\mathbf{0}$ over the interface $\mathbb{I} = \{\{a\}, \{b\}\}$. Note that $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$ because any uncoordinated observer that passes p is happy regardless of whether a and b are executed. Moreover, such observer would be unable to detect if both a and b are executed because those actions take place over different parts of the interface. Hence, $\mathsf{str}(q) \not\subseteq \mathsf{str}(p)$ because $a.b \in \mathsf{str}(q)$ but $a.b \notin \mathsf{str}(p)$ despite $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$. Our notion of reduction w.r.t a Mazurkiewicz trace will account for such mismatch and, e.g., $p \stackrel{[a.b]_D}{\Longrightarrow} \mathbf{0}$ will hold. Intuitively, the relation $\stackrel{[s]_D}{\Longrightarrow}$ accounts for reductions in which s may be partially executed over some of the parts of the interface (but complete in at least one of them).

We write $s <_D t$ iff $t \in [ss']_D$ for some $s' \neq \epsilon$, i.e., t extends s (up-to the swapping of independent actions), and $s \leq_D t$ stands for $s <_D t$ or $s \equiv_D t$. Then, the set of maximal reductions of p within a trace $[s]_D$ is

$$\mathsf{str}(p, [s]_D) = \max_{<_D}\{t \mid t \in \mathsf{str}(p), t \leq_D s\}$$

where $\max_{<_D}$ denotes the maximal elements of a set according to the order $<_D$. If $t \in \mathsf{str}(p, [s]_D)$ then t cannot be extended with any symbol a such that $t' \equiv_D ta \in \mathsf{str}(p, [s]_D)$ and $p \stackrel{t'}{\Longrightarrow}$.

The following last ingredient allows us to ensure that different maximal prefixes in $\mathsf{str}(p, [s]_D)$ actually execute the actions in $[s]_D$ (although some prefixes can be partial). We recall that $\upharpoonright$ stands for the operation that projects a string over an alphabet. Let $\mathbb{I}$ be the interface that induces the dependency relation D, we write $\mathsf{str}(p, [s]_D)^\dagger$ if $\mathsf{str}(p, [s]_D)$ *jointly-completes* $[s]_D$, which is defined by

$$\mathsf{str}(p, [s]_D)^\dagger \quad \iff \quad \forall I \in \mathbb{I}. \exists t \in \mathsf{str}(p, [s]_D). s \upharpoonright I = t \upharpoonright$$

Example 3.8. Consider the processes $p = \tau.a.\mathbf{0} + \tau.b.\mathbf{0}$ and $q = a.b.\mathbf{0}$ and the interface $\mathbb{I} = \{\{a\},\{b\}\}$. Let D be the dependency relation induced by $\mathbb{I}$. Then, we have

$$\begin{array}{rl} \mathsf{str}(p) = & \{\epsilon, a, b\} \\ \mathsf{str}(q) = & \{\epsilon, a, b, a.b\} \end{array}$$

The restriction of $<_D$ to the elements in $\mathsf{str}(q)$ is

$$\{(\epsilon, a), (\epsilon, b), (a, a.b), (b, a.b)\}$$

Note that $b <_D ab$ because b can be extended with a and $ab \in [ba]_D$. Then,

$$\begin{array}{rl} \mathsf{str}(p,[a.b]_D) = & \max_{<_D}\{t \mid t \in \mathsf{str}(p), t \leq_D a.b\} = \max_{<_D}\{\epsilon, a, b\} = \{a, b\} \\ \mathsf{str}(q,[a.b]_D) = & \max_{<_D}\{t \mid t \in \mathsf{str}(q), t \leq_D a.b\} = \max_{<_D}\{\epsilon, a, b, a.b\} = \{a.b\} \end{array}$$

Both cases jointly-completes $[a.b]_D$, i.e., $\mathsf{str}(p,[a.b]_D)^\dagger$ and $\mathsf{str}(q,[a.b]_D)^\dagger$ do hold.

On the contrary, if we consider $r = a.\mathbf{0}$ we have that $\mathsf{str}(r,[a.b]_D) = \{a\}$, which does not jointly-complete $[a.b]_D$, because none of the strings in $\mathsf{str}(r,[a.b]_D)$ matches b. □

Let D be the dependency relation induced by the interface $\mathbb{I}$. We let

- $p \overset{[s]_D}{\Longrightarrow} p'$ if and only if $\mathsf{str}(p,[s]_D)^\dagger$ and $\exists s' \in \mathsf{str}(p,[s]_D)$ such that $p \overset{s'}{\Longrightarrow} p'$.
- $p \Downarrow [s]_D$ if $\forall s' \in [s]_D$ then $p \Downarrow s'$.
- $(p \text{ after } [s]_D) = \{p' \mid p \overset{[s]_D}{\Longrightarrow} p'\}$.

We adopt usual conventions for abbreviating notation when dealing with transition relations and write, e.g., $p \overset{[s]_D}{\Longrightarrow}$ instead of *there exists* p' *such that* $p \overset{[s]_D}{\Longrightarrow} p'$. Analogously, $p \overset{[s]_D}{\not\Longrightarrow}$ stands for *there does not exist* p' *such that* $p \overset{[s]_D}{\Longrightarrow} p'$.

In the definition below, the condition $L \subseteq I$ with $I \in \mathbb{I}$ captures the idea that each observation is relative to a specific part of the interface.

Definition 3.9. Let $\mathbb{I}$ be an interface and D the dependency relation induced by $\mathbb{I}$. Then, $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$ if for every $s \in \mathsf{Act}^*$, for any part $I \in \mathbb{I}$, for all finite $L \subseteq I$, if $p \Downarrow [s]_D$ then

(1) $q \Downarrow [s]_D$.
(2) $(p \text{ after } [s]_D)$ MUST L implies $(q \text{ after } [s]_D)$ MUST L.

The following three lemmata are instrumental to the proof of the correspondence theorem and characterise the relation between the Mazurkiewicz traces of related processes.

Lemma 3.10. *Let $\mathbb{I}$ be an interface and D the dependency relation induced by $\mathbb{I}$. If $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$ then for all $s \in \mathsf{Act}^*$ we have that $p \Downarrow [s]_D$ implies*

(1) $q \Downarrow [s]_D$.
(2) $s \in \mathsf{str}(q)$ *implies that* $p \overset{[s]_D}{\Longrightarrow}$.

Proof. Assume $\mathbb{I} = \{I_i\}_{i \in 0\ldots n}$ and D the induced dependency relation.

(1) By contradiction. Suppose there exists $s = a_1 \ldots a_m$ such that $p \Downarrow [s]_D$ and $q \Uparrow [s]_D$. Then, take the observer $o = \Pi_{i \in 0,\ldots n} o_i$ with o_i defined as follows

$$o_i = \tau.\mathbf{1} + \overline{b^i_1}.(\tau.\mathbf{1} + \ldots (\tau.\mathbf{1} + \overline{b^i_{k_i}}.\tau.\mathbf{1})\ldots) \text{ with } s \upharpoonright I_i = b^i_1 \ldots b^i_{k_i}$$

For each maximal computation of $p \parallel o$, we proceed by unzipping the computation to conclude that $o \overset{\bar{t}}{\Longrightarrow}$ and $p \overset{t}{\Longrightarrow}$ for some t. Note that $o \Downarrow t$ for all $t \in \mathsf{Act}^*$ and $o \overset{\bar{t}}{\Longrightarrow}$ implies that there exists t' such that $\bar{t}t' \in [\bar{s}]_D$. Since $p \Downarrow [s]_D$, we have $p \Downarrow t$.

Consequently, every maximal computation of $p \parallel o$ is finite, i.e., $p \parallel o \Longrightarrow p' \parallel o' \not\rightarrow$. By induction on the length of t, it follows that $o \stackrel{t}{\Longrightarrow} o' \not\rightarrow$ implies $o' = \Pi_{i\in 0,\ldots n}\mathbf{1}$. Consequently, every maximal computation of $p \parallel o$ is observer-successful and p must o.

Since $q \Uparrow [s]_D$ there exists $tt' \in [s]_D$ such that $q \stackrel{t}{\Longrightarrow} q_0$ and $q_0 \Uparrow$. By induction on the length of $\bar{t}$ it follows that there exists o' such that $o \stackrel{\bar{t}}{\Longrightarrow} o'$ and $o' = \Pi_{i\in 0,\ldots n} o_i''$ where $o_i'' = \tau.\mathbf{1}$ or $o_i'' = (\tau.\mathbf{1} + a.(\ldots))$ for all i. Then, there exists a maximal (divergent) observer-unsuccessful computation $q \parallel o \rightarrow q_0 \parallel o' \rightarrow q_1 \parallel o' \rightarrow \ldots \rightarrow q_j \parallel o' \rightarrow \ldots$. Consequently, q must o, which contradicts the assumption $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$.

(2) By contradiction. Suppose that there exists $s = a_1 \ldots a_m$ such that $p \Downarrow [s]_D$, $s \in \mathsf{str}(q)$ and $p \stackrel{[s]_D}{\not\Longrightarrow}$. From $p \stackrel{[s]_D}{\not\Longrightarrow}$, either (i) $(\mathsf{str}(p,[s]_D))^\dagger$ does not hold; or (ii) $\nexists s' \in \mathsf{str}(p,[s]_D)$ such that $p \stackrel{s'}{\Longrightarrow}$. Note that (ii) is impossible: because $\epsilon \in \mathsf{str}(p)$ for all p and $\epsilon \leq_D s$ for all s and D. Consequently, for all p, s and D we have $\{t \mid t \in \mathsf{str}(p), t \leq_D s\} \neq \emptyset$ and $\mathsf{str}(p,[s]_D) \neq \emptyset$. By the definition of $\mathsf{str}(p,[s]_D)$, $s' \in \mathsf{str}(p,[s]_D)$ implies $s' \in \mathsf{str}(p)$, and therefore $p \stackrel{s'}{\Longrightarrow}$, which contradicts (ii). Then, (i) holds. Consequently,

$$\exists I_j \in \mathbb{I}.\forall t \in (\mathsf{str}(p,[s]_D)).s \upharpoonright I_j \neq t \upharpoonright I_j \tag{3.1}$$

Then, choose the observer $o = \Pi_{i\in 0,\ldots n} o_i$ with o_i defined as follows

$$o_i = \tau.\mathbf{1} + \overline{b_1^i}.(\tau.\mathbf{1} + \ldots (\tau.\mathbf{1} + \overline{b_{k_i}^i}.\mathbf{1})\ldots) \qquad \text{if } b_{k_i}^i \neq a_m$$
$$o_i = \tau.\mathbf{1} + \overline{b_1^i}.(\tau.\mathbf{1} + \ldots (\tau.\mathbf{1} + \overline{b_{k_i}^i}.\mathbf{0})\ldots) \qquad \text{if } b_{k_i}^i = a_m$$

with $s \upharpoonright I_i = b_1^i \ldots b_{k_i}^i$.

Since $s \in \mathsf{str}(q)$, $q \stackrel{s}{\Longrightarrow} q'$. By (1) above, $p \Downarrow [s]_D$ implies $q \Downarrow [s]_D$. Consequently, $q' \Downarrow$. Then, there is a computation $q \stackrel{s}{\Longrightarrow} q' \Longrightarrow q'' \not\rightarrow$. Moreover, we can build an unsuccessful computation of $o \stackrel{\bar{s}}{\Longrightarrow} o'' = \Pi_{i\in 0,\ldots n} o_i'' \not\rightarrow$ where $o_j'' = \mathbf{0}$ for $j \in 0,\ldots,n$ and $b_{k_j}^j = a_n$. By zipping the computations $q \stackrel{s}{\Longrightarrow} q''$ and $o \stackrel{\bar{s}}{\Longrightarrow} o''$, we obtain a maximal computation of $q \parallel o$ that is observer-unsuccessful. Consequently, q must o.

Take a maximal computation of $p \parallel o \Longrightarrow$. By unzipping it, $p \stackrel{t}{\Longrightarrow} p'$ and $o \stackrel{\bar{t}}{\Longrightarrow} o'$. By construction of o, $o \stackrel{s}{\Longrightarrow}$. By Lemma 3.6, $o \stackrel{r}{\Longrightarrow}$ iff $r \in [s]_D$. Then, $o \stackrel{\bar{t}}{\Longrightarrow} o'$ implies that there exists t' such that $\bar{t}t' \in [\bar{s}]_D$. From $p \Downarrow [s]_D$, we have $p' \Downarrow$ and, consequently, $p' \not\rightarrow$. Also $o \Downarrow \bar{t}$ holds because o is finite; moreover, $o' \not\rightarrow$. Hence, $p \parallel o \Longrightarrow p' \parallel o'$ is a finite maximal computation.

By assumption, $[s]_D \cap \mathsf{str}(p) = \emptyset$ holds (otherwise, $(\mathsf{str}(p,[s]_D))^\dagger$ should hold). Therefore $t \notin [s]_D$. Then, t is a prefix of a string in $[s]_D$, i.e., there exists a and t'' such that $tt''a \in [s]_D$ and for all p' if $p \stackrel{t}{\Longrightarrow} p'$ then $p' \stackrel{[t''a]_D}{\not\Longrightarrow}$. Without lost of generality we assume $a = a_n$ (otherwise, the definition of o can be changed accordingly). By construction, $o \stackrel{\bar{t}}{\Longrightarrow} o'$ implies $o' = \Pi_{i\in 0,\ldots n} o_i'$ and $o' \stackrel{[t''a]_D}{\Longrightarrow}$ where for any i either (a) $o_i' = \tau.1 + \bar{b}.(\ldots)$ or (b) $o_i' = \mathbf{1}$. Case (a) is not possible, because we assume $o' \not\rightarrow$. For case (b), we proceed by zipping the computations $p \stackrel{t'}{\Longrightarrow} p'' \not\rightarrow$ and $o \stackrel{\overline{t'}}{\Longrightarrow} o' \not\rightarrow$, which is observer-successful. Consequently, every maximal computation of $p \parallel o$ is observer-successful and p must o, which is in contradiction with the assumption $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$. □

Lemma 3.11. *If* $(p$ after $[s]_D)$ must L *for some finite* $L \subseteq \mathsf{Act}$, *then* $p \stackrel{[s]_D}{\Longrightarrow}$.

Proof. Suppose $p \stackrel{[s]_D}{\not\Longrightarrow}$. Then $(p \text{ after } [s]_D) = \emptyset$ and, by definition, $\emptyset \text{ MUST } L$ for every finite $L \subseteq \mathsf{Act}$. □

Lemma 3.12. *If $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, $s \in \mathsf{str}(q)$ and $p \Downarrow [s]_D$ then $p \stackrel{[s]_D}{\Longrightarrow}$.*

Proof. Assume that $p \stackrel{[s]_D}{\not\Longrightarrow}$. Then, $(p \text{ after } [s]_D) = \emptyset$. Hence, $(p \text{ after } [s]_D) \text{ MUST } L$ for every finite $L \subseteq A$. Since $p \Downarrow [s]_D$ and $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, $q \Downarrow [s]_D$. By straightforward induction on the length of the reduction, we can show that $q \stackrel{t}{\Longrightarrow} q'$ implies $\mathtt{n}(q') \subseteq \mathtt{n}(q)$ for all t (i.e., the names $\mathtt{n}(q')$ of q' are included in the names of q). Moreover, $\mathsf{init}(q) \subseteq \mathtt{n}(q)$ trivially holds. Consequently, $\bigcup\{\mathsf{init}(q') \mid q \stackrel{[s]_D}{\Longrightarrow} q'\} \subseteq \mathtt{n}(q)$. Since, $\mathtt{n}(q)$ is finite, we can conclude that the set $\bigcup\{\mathsf{init}(q') \mid q \stackrel{[s]_D}{\Longrightarrow} q'\}$ is finite. Therefore, we can find an action a such that $q \stackrel{sa}{\not\Longrightarrow}$. Then $(q \text{ after } [s]_D) \text{ MUST } \{a\}$ while $(p \text{ after } [s]_D) \text{ MUST } \{a\}$, which contradicts the hypothesis $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$. □

Theorem 3.13. $\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} = \preceq^{\mathbb{I}}_{\mathsf{unc}}$.

Proof.

(⊆) Actually we prove that $p \not\preceq^{\mathbb{I}}_{\mathsf{unc}} q$ implies $p \not\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$. Let D be the dependency relation induced by $\mathbb{I}$. Assume that there exists $s = a_1 \dots a_n$ and $I_j \in \mathbb{I}$ and $L \subseteq I_j$ such that
(1) $p \Downarrow [s]_D$ and $q \Uparrow [s]_D$, or
(2) $s \in \mathsf{str}(q)$ and $\forall t \in [s]_D . t \notin \mathsf{str}(p)$ or
(3) $(p \text{ after } [s]_D) \text{ MUST } L$ and $(q \text{ after } [s]_D) \text{ M}\not\text{UST } L$
For each case we show that there exists an observer such that $p \text{ must } o$ and $q \text{ m}\not\text{ust } o$. For the two first cases, we take the observers as defined in proof of Lemma 3.10. For the third one, we take $o = \Pi_{i \in 0,\dots n} o_i$ with o_i defined as follows

$$o_i = \tau.\mathbf{1} + b_1.(\tau.\mathbf{1} + \dots (\tau.\mathbf{1} + b_k.\mathbf{1}) \dots) \qquad \text{if } i \neq j$$
$$o_i = \tau.\mathbf{1} + b_1.(\tau.\mathbf{1} + \dots (\tau.\mathbf{1} + b_k.\textstyle\sum_{a \in L} a.\mathbf{1}) \dots) \qquad \text{if } i = j$$

with $s \upharpoonright I_i = b_1 \dots b_k$.

(⊇) We prove $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$ implies $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$. Actually, the proof follows by showing that $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$ and $q \text{ m}\not\text{ust } o$ imply $p \text{ m}\not\text{ust } o$. Assume there exists an unsuccessful computation

$$q \parallel o = q_0 \parallel o_0 \stackrel{\tau}{\to} \dots \stackrel{\tau}{\to} q_k \parallel o_k \stackrel{\tau}{\to} \dots$$

Consider the following cases:

(1) **The computation is finite**, i.e., there exists n such that $q_n \parallel o_n \stackrel{\mathcal{I}}{\not\to}$.
By unzipping the computation we have $q_0 \stackrel{s}{\Longrightarrow} q_n$ and $o_0 \stackrel{\bar{s}}{\Longrightarrow} o_n$, which is unsuccessful, i.e., $o_i \stackrel{\checkmark}{\not\to}$ for all $0 \leq i \leq n$.
Moreover, $q_n \text{ M}\not\text{UST } \mathsf{init}(o_n)$. Hence $(q \text{ after } [s]_D) \text{ M}\not\text{UST } \mathsf{init}(o_n)$.
   (a) Case $p \Uparrow [s]_D$, i.e., $\exists t \in [s]_D$ and $p \Uparrow t$. By Corollary 3.7 (4), $o \stackrel{\bar{s}}{\Longrightarrow}$ implies $o \stackrel{\bar{t}}{\Longrightarrow}$ also unsuccessful, and hence there is an unsuccessful computation of $p \parallel o$.
   (b) Case $p \Downarrow [s]_D$. Note that $s \in \mathsf{str}(q)$. By Lemma 3.10 (2), $\exists t \in [s]_D : t \in \mathsf{str}(p)$. Hence, $(p \text{ after } [s]_D) \neq \emptyset$. Since $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, $(q \text{ after } [s]_D) \text{ M}\not\text{UST } \mathsf{init}(o_n)$ implies $(p \text{ after } [s]_D) \text{ M}\not\text{UST } \mathsf{init}(o_n)$. Therefore, exists some $p' \in (p \text{ after } [s]_D)$ and $p' \text{ M}\not\text{UST } \mathsf{init}(o_n)$. Hence, $\exists t' \in [s]_D . p \stackrel{t'}{\Longrightarrow}$. By Corollary 3.7 (4), $o \stackrel{\bar{t}'}{\Longrightarrow}$ unsuccessful, and hence there is an unsuccessful computation of $p \parallel o$.

(2) **The computation is infinite**. We consider two cases:
   (a) There exist $s \in \mathsf{str}(q)$ and $\overline{s} \in \mathsf{str}(o)$ such that $q \Uparrow s$ or $o \Uparrow \overline{s}$. We proceed by case analysis.
      (i) $q \Uparrow [s]_D$: Since $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, $p \Uparrow [s]_D$. Therefore, $\exists t \in [s]_D$ such that $p \Uparrow t$. By Corollary 3.7 (4), $o \stackrel{\overline{t}}{\Longrightarrow}$ unsuccessful, and hence there is an unsuccessful computation of $p \parallel o$.
      (ii) $q \Downarrow [s]_D$ (and $o \Uparrow \overline{s}$): By Lemma 3.12, $\exists t \in [s]_D : t \in \mathsf{str}(p)$. By Corollary 3.7 (2), $o \Uparrow \overline{t}$, and hence there is an unsuccessful computation of $p \parallel o$.
   (b) $\forall n. q_n \Downarrow$ and $o_n \Downarrow$. For every n, take $s \in \mathsf{Act}^*$ such that $q \stackrel{s}{\Longrightarrow} q_n$ and $q \Downarrow s$ (this is possible because $q \parallel o \Longrightarrow q_n \parallel o_n$ is unsuccessful and $\forall i \leq n. q_i \Downarrow$). By Lemma 3.10, $q \stackrel{s}{\Longrightarrow} q_n$ and $q \Downarrow s$ implies either (i) $p \Uparrow [s]_D$ or (ii) $p \Downarrow [s]_D$ and $p \stackrel{[s]_D}{\Longrightarrow}$.
      (i) $p \Uparrow [s]_D$: $\exists t \in [s]_D$ such that $p \Uparrow t$. By Corollary 3.7 (4), $o \stackrel{\overline{s}}{\Longrightarrow}$ unsuccessful implies $o \stackrel{\overline{t}}{\Longrightarrow}$ unsuccessful, and hence there is an unsuccessful computation of $p \parallel o$.
      (ii) $p \Downarrow [s]_D$ and $p \stackrel{[s]_D}{\Longrightarrow}$. Since $q_n \Downarrow$, there exists q_m such that $q_n \Longrightarrow q_m \not\rightarrow$ and $q_m \stackrel{a}{\rightarrow} q_{m+1}$. Consequently, $(q \text{ after } [s]_D)$ MUST L for all L such that $a \notin L$. Since $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, then for all L such that $a \notin L$, we have $(p \text{ after } [s]_D)$ MUST L. Moreover, $o_n \Longrightarrow o_m \stackrel{\overline{a}}{\rightarrow} o_{m+1}$. Therefore, there exists $p_n \in (p \text{ after } [s]_D)$ and $p_n \stackrel{a}{\Longrightarrow} p_{m+1}$. Hence, for all n there is an unsuccessful computation $p \parallel o \Longrightarrow p_n \parallel o_n \Longrightarrow p_{m+1} \parallel o_{m+1}$. □

In the following we will write $L^I_{p,[s]_D}$ for the smallest set such that $(p \text{ after } [s]_D)$ MUST L and $L \subseteq I$ imply $L^I_{p,[s]_D} \subseteq L$.

Example 3.14. We take advantage of the alternative characterisation of the uncoordinated preorder to show that the three processes for the broker in Example 3.1 are equivalent when considering $\mathbb{I} = \{\{req\}, \{reqF\}, \{reqH\}\}$. Actually, we will only consider $B_0 \approx^{\mathbb{I}}_{\mathsf{unc}} B_1$, as the proofs for $B_0 \approx^{\mathbb{I}}_{\mathsf{unc}} B_2$ and $B_1 \approx^{\mathbb{I}}_{\mathsf{unc}} B_2$ are analogous.

Firstly, we have to consider that $B_0 \Downarrow s$ and $B_1 \Downarrow s$ for any s because B_0 and B_1 do not have infinite computations. The relation between must-sets are described in the two tables below. The first table shows the sets $(B_0 \text{ after } [s]_D)$ and $L^I_{B_0,[s]_D}$. Note that $[s]_D$ in the first column will be represented by any string $s' \in [s]_D$. Moreover, we write "$-$" in the three last columns whenever $L^I_{B_0,[s]_D}$ does not exist. The second table does the same for B_1. In the tables, we let B'_0 stand for $\tau.\overline{reqF}.\mathbf{0} + \tau.\overline{reqH}.\mathbf{0} + \tau.\overline{reqH}.\overline{reqF}.\mathbf{0}$ and B'_1 stand for $\tau.\overline{reqF}.\mathbf{0} + \tau.\overline{reqH}.\mathbf{0} + \tau.\overline{reqF}.\overline{reqH}.\mathbf{0}$.

$[s]_D$	B_0 after $[s]_D$	$L^{\{req\}}_{B_0,[s]_D}$	$L^{\{reqH\}}_{B_0,[s]_D}$	$L^{\{reqF\}}_{B_0,[s]_D}$
ϵ	B_0	$\{req\}$	$-$	$-$
req	$\{B'_0, \overline{reqF}.\mathbf{0}, \overline{reqH}.\mathbf{0}, \overline{reqH}.\overline{reqF}.\mathbf{0}\}$	$-$	$-$	$-$
$req.\overline{reqF}$	$\{0\}$	$-$	$-$	$-$
$req.\overline{reqH}$	$\{0, \overline{reqF}.\mathbf{0}\}$	$-$	$-$	$-$
$req.\overline{reqF}.\overline{reqH}$	$\{0\}$	$-$	$-$	$-$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$[s]_D$	B_1 after $[s]_D$	$L^{\{req\}}_{B_0,[s]_D}$	$L^{\{reqH\}}_{B_0,[s]_D}$	$L^{\{reqF\}}_{B_0,[s]_D}$
ϵ	B_1	$\{req\}$	–	–
req	$\{B'_1, \overline{reqF}.\mathbf{0}, \overline{reqH}.\mathbf{0}, \overline{reqF}.\overline{reqH}.\mathbf{0}\}$	–	–	–
$req.\overline{reqF}$	$\{0, \overline{reqH}\}$	–	–	–
$req.\overline{reqH}$	$\{0\}$	–	–	–
$req.\overline{reqF}.\overline{reqH}$	$\{0\}$	–	–	–
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

By inspecting the tables, we can check that for any possible trace $[s]_D$ and $I \in \mathbb{I}$, it holds that $L^I_{B_0,[s]_D} = L^I_{B_1,[s]_D}$. Consequently, $(B_0$ after $[s]_D)$ MUST L iff $(B_1$ after $[s]_D)$ MUST L and thus we have $B_0 \approx^{\mathbb{I}}_{\mathsf{unc}} B_1$. □

We now present two additional examples that help us understand the discriminating capability of the uncoordinated preorder and its differences with the classical must preorder.

The first of these examples shows that a process that does not communicate its internal choices over all parts of its interface is useless in a distributed context.

Example 3.15. Consider the process $p = \tau.a.\mathbf{0} + \tau.b.\mathbf{0}$ that is intended to be used by two processes with the following interface: $\mathbb{I} = \{\{a\}, \{b\}\}$. We show that this process is less useful than 0 in an uncoordinated context, i.e., $\tau.a.\mathbf{0} + \tau.b.\mathbf{0} \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} 0$. It is immediate to see that p and 0 strongly converge for any $s \in \mathsf{Act}^*$, then the minimal sets $L^{\{a\}}_{p,[s]_D}$, $L^{\{b\}}_{p,[s]_D}$, $L^{\{a\}}_{0,[s]_D}$ and $L^{\{b\}}_{0,[s]_D}$ presented in the tables below are sufficient for proving our claim.

$[s]_D$	p after $[s]_D$	$L^{\{a\}}_{p,[s]_D}$	$L^{\{b\}}_{p,[s]_D}$
ϵ	p, a, b	–	–
a	$\{0\}$	–	–
b	$\{0\}$	–	–
other	$\emptyset$	$\emptyset$	$\emptyset$

$[s]_D$	0 after $[s]_D$	$L^{\{a\}}_{0,[s]_D}$	$L^{\{b\}}_{0,[s]_D}$
ϵ	0	–	–
a	$\emptyset$	$\emptyset$	$\emptyset$
b	$\emptyset$	$\emptyset$	$\emptyset$
other	$\emptyset$	$\emptyset$	$\emptyset$

Note that differently from the classical must preorder, the uncoordinated preorder does not consider the must-set $\{a, b\}$ to distinguish p from 0 because this set involves channels in different parts of the interface. The key point here is that each internal reduction of p is observed just by one part of the interface: the choice of branch a is only observed by one process and the choice of b is observed by the other one. Since uncoordinated observers do not intercommunicate, they can only report success simultaneously if they can do it independently from the interactions with the tested process, but such observers are exactly the ones that 0 can pass.

Like in the classical must preorder, we have that $0 \not\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \tau.a.\mathbf{0} + \tau.b.\mathbf{0}$. This is witnessed by the observer $o = \overline{a}.\mathbf{0} + \tau.\mathbf{1} \parallel \mathbf{1}$ that is passed by 0 but not by $\tau.a.\mathbf{0} + \tau.b.\mathbf{0}$. □

The second example shows that the uncoordinated preorder falls somehow short with respect to the target we set in the introduction of allowing processes to swap actions that are targeted to different partners.

Example 3.16. Consider the interface $\mathbb{I} = \{\{a\}, \{b\}\}$ and the two pairs of processes

- $a.b.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$ and $b.a.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$.
- $a.b.\mathbf{0}$ and $b.a.\mathbf{0}$.

By inspecting traces and must-sets in the two tables below, where we use p and q to denote $a.b.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$ and $b.a.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$

$[s]_D$	p after $[s]_D$	$L^{\{a\}}_{p,[s]_D}$	$L^{\{b\}}_{p,[s]_D}$
ϵ	$\{p\}$	$\{a\}$	$\{b\}$
a	$\{b.\mathbf{0}, 0\}$	$-$	$-$
b	$\{0\}$	$-$	$-$
ab	$\{0\}$	$-$	$-$
other	$\emptyset$	$\emptyset$	$\emptyset$

$[s]_D$	q after $[s]_D$	$L^{\{a\}}_{q,[s]_D}$	$L^{\{b\}}_{q,[s]_D}$
ϵ	$\{p\}$	$\{a\}$	$\{b\}$
a	$\{0\}$	$-$	$-$
b	$\{a.\mathbf{0}, 0\}$	$-$	$-$
ab	$\{0\}$	$-$	$-$
other	$\emptyset$	$\emptyset$	$\emptyset$

it is easy to see that

$$a.b.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0} \approx^{\mathbb{I}}_{\mathsf{unc}} b.a.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$$

However, by using $o = \overline{a}.\mathbf{1} \parallel \mathbf{1}$ and $o' = \mathbf{1} \parallel \overline{b}.\mathbf{1}$ as observers, it can be shown that

$$a.b.\mathbf{0} \not\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} b.a.\mathbf{0} \qquad \text{and} \qquad b.a.\mathbf{0} \not\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} a.b.\mathbf{0}$$

Note that $o = \overline{a}.\mathbf{1} \parallel \mathbf{1}$ actually interacts with the process under test by using just one part of the interface and relies on the fact that the remaining part of the interface stays idle. Thanks to this ability, uncoordinated observers have still a limited power to track some dependencies among actions on different parts of the interface.

The preorder presented in the next section limits further the discriminating power of observers and allows us to equate processes $a.b.\mathbf{0}$ and $b.a.\mathbf{0}$. □

4. A testing preorder with individualistic observers

In this section we explore a notion of equivalence equating processes that can freely permute actions over different parts of their interfaces. As for the uncoordinated observers, the targeted scenario is that of a service with a partitioned interface interacting with two or more independent processes by using separate sets of ports. In addition, each component of an observer cannot exploit any knowledge about the design choices made by the other components, i.e., each of them has a local view of the behaviour of the process that ignores all actions controlled by the remaining components. Local views are characterised in terms of a projection operator defined as follows.

Definition 4.1 (Projection)**.** Let $V \subseteq \mathcal{N}$ be a set of observable ports. We write $p \upharpoonright V$ for the process obtained by hiding all actions of p over channels that are not in V. Formally,

$$\frac{p \xrightarrow{\alpha} p' \quad \alpha \in V \cup \overline{V}}{p \upharpoonright V \xrightarrow{\alpha} p' \upharpoonright V} \qquad \frac{p \xrightarrow{\alpha} p' \quad \alpha \notin V \cup \overline{V}}{p \upharpoonright V \xrightarrow{\tau} p' \upharpoonright V}$$

Example 4.2. Let $p = \overline{a}.p_1 + b.p_2$ be a process. Note that $p \xrightarrow{\overline{a}} p_1$ and $p \xrightarrow{b} p_2$. Then, the projection of p over the channel a, i.e., $p \upharpoonright \{a\}$, has the following two transitions: $p \upharpoonright \{a\} \xrightarrow{\overline{a}} p_1 \upharpoonright \{a\}$ and $p \upharpoonright \{a\} \xrightarrow{\tau} p_1 \upharpoonright \{a\}$ where the action $\overline{a}$ of p over the visible channel a is reflected on the label of the transition of $p \upharpoonright \{a\}$ while the action over the non-visible channel b is taken as an internal action. □

Definition 4.3 (Individualistic (must) preorder $\sqsubseteq^{\mathbb{I}}_{\mathsf{ind}}$)**.** Let $\mathbb{I} = \{I_i\}_{i \in 0..n}$ be an interface. We say $p \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} q$ iff for all uncoordinated observer $o = \Pi_{i \in 0..n} o_i$, for all $i \in 0..n$, $p \upharpoonright I_i$ must o_i implies $q \upharpoonright I_i$ must o_i.

Note that $a.b.\mathbf{0}$ and $b.a.\mathbf{0}$ cannot be distinguished anymore by the observer $o = \overline{a}.\mathbf{1} \parallel \mathbf{1}$ used in the previous section to prove $a.b.\mathbf{0} \not\sqsubseteq_{\mathsf{unc}}^{\{\{a\},\{b\}\}} b.a.\mathbf{0}$ (Example 3.16), because $a.b.\mathbf{0} \upharpoonright \{a\}$ must $\overline{a}.\mathbf{1}$, $b.a.\mathbf{0} \upharpoonright \{a\}$ must $\overline{a}.\mathbf{1}$, $a.b.\mathbf{0} \upharpoonright \{b\}$ must $\mathbf{1}$ and $b.a.\mathbf{0} \upharpoonright \{b\}$ must $\mathbf{1}$. Indeed, later (Example 4.11) we will see that:

$$a.b.\mathbf{0} \approx_{\mathsf{ind}}^{\{\{a\},\{b\}\}} b.a.\mathbf{0}.$$

4.1. **Semantic characterisation.** In this section we address the characterisation of the individualistic preorder in terms of traces. We start by introducing an equivalence relation over traces that ignores hidden actions.

Definition 4.4 (Filtered traces)**.** Let $I \subseteq \mathsf{Act}$. Two strings $s, t \in \mathsf{Act}^*$ are *equivalent up-to* I, written $s \overset{\bullet}{\equiv}_I t$, if and only if $s \upharpoonright I = t \upharpoonright I$. We write $[[s]]_I$ for the equivalence class of s.

Basically, two traces are equivalent up-to I when they coincide after the removal of hidden actions. For instance, $aa \overset{\bullet}{\equiv}_{\{a\}} aba \overset{\bullet}{\equiv}_{\{a\}} ababbb \overset{\bullet}{\equiv}_{\{a\}} \ldots$.

As for the distributed preorder, we extend the notions of reduction, convergence and residuals to equivalence classes of filtered traces.

- $q \overset{[[s]]_I}{\Longrightarrow} q'$ if and only if $\exists t \in [[s]]_I$ such that $q \overset{t}{\Longrightarrow} q'$.
- $p \Downarrow [[s]]_I$ if and only if $\forall t \in [[s]]_I . p \Downarrow t$.
- $(p \text{ after } [[s]]_I) = \{p' \mid p \overset{[[s]]_I}{\Longrightarrow} p'\}$.

The following auxiliary result establishes properties relating reductions, hiding and filtered traces, which will be useful in the proof of the correspondence theorem.

Lemma 4.5.

(1) $p \overset{s}{\Longrightarrow} p'$ *implies* $p \upharpoonright I \overset{s \upharpoonright I}{\Longrightarrow} p' \upharpoonright I$.
(2) $p \upharpoonright I \overset{s}{\Longrightarrow} p' \upharpoonright I$ *implies* $\exists t \in [[s]]_I$ *and* $p \overset{t}{\Longrightarrow} p'$.
(3) $p \Uparrow [[s]]_I$ *implies* $p \upharpoonright I \Uparrow s \upharpoonright I$.
(4) $(p \text{ after } [[s]]_I)$ MUST L *iff* $(p \upharpoonright I \text{ after } s \upharpoonright I)$ MUST $L \cap I$.

Proof. The proof follows by induction on the length of s. □

The alternative characterisation for the individualistic preorder is given in terms of filtered traces.

Definition 4.6. Let $p \preceq_{\mathsf{ind}}^{\mathbb{I}} q$ if for every $I \in \mathbb{I}$, for every $s \in I^*$, and for all finite $L \subseteq I$, if $p \Downarrow [[s]]_I$ then

(1) $q \Downarrow [[s]]_I$
(2) q $(p \text{ after } [[s]]_I)$ MUST $L \cup (\mathsf{Act} \backslash I)$ implies $(q \text{ after } [[s]]_I)$ MUST $L \cup (\mathsf{Act} \backslash I)$

We would like to draw attention to condition 2 above; it only considers must-sets that always include all the actions in $(\mathsf{Act} \backslash I)$ to avoid the possibility of distinguishing reachable states because of actions that are not in I. Consider that this condition could be formulated as follows: for all finite $L \subseteq \mathsf{Act}$,

$(p \text{ after } [[s]]_I)$ MUST L implies $\exists L'$ such that $(q \text{ after } [[s]]_I)$ MUST L' and $L \cap I = L' \cap I$

which makes evident that only the actions from the observable part of the interface are relevant. We adopted the first formulation because it resembles the original characterisation of the must preorder.

The following lemmata are analogous to those for the uncoordinated preorder and their proof follows similarly (proof details are in Appendix A).

Lemma 4.7. *If* $p \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} q$ *then for all* $s \in \mathsf{Act}^*$ *and* $I \in \mathbb{I}$*, we have that* $p \Downarrow [[s]]_I$ *implies*
(1) $q \Downarrow [[s]]_I$
(2) $s \in \mathsf{str}(q)$ *implies that there exists* $t \in [[s]]_I$ *such that* $t \in \mathsf{str}(p)$.

Lemma 4.8. *if* $(p \text{ after } [[s]]_I)$ MUST L *for some* $L \subseteq \mathsf{Act}$*, then* $\exists t \in [[s]]_I : t \in \mathsf{str}(p)$.

We rely on the following auxiliary results relating the traces of processes in the must preorders.

Lemma 4.9. *If* $p \preceq^{\mathbb{I}}_{\mathsf{ind}} q$*,* $s \in \mathsf{str}(q)$ *and* $p \Downarrow [[s]]_I$ *with* $I \in \mathbb{I}$ *then* $t \in ([[s]]_I \cap \mathsf{str}(p))$.

Theorem 4.10. $\sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} = \preceq^{\mathbb{I}}_{\mathsf{ind}}$.

Proof. The proof follows along the lines of that of Theorem 3.13 (see details in Appendix A). □

Example 4.11. Consider the processes $p = a.b.\mathbf{0}$ and $q = b.a.\mathbf{0}$ and the interface $\mathbb{I} = \{\{a\}, \{b\}\}$. The table below shows the analysis for the part of the interface $\{a\}$.

$[[s]]_{\{a\}}$	p after $[[s]]_{\{a\}}$	$L^{\{a\}}_{p,[[s]]_I}$	q after $[[s]]_{\{a\}}$	$L^{\{a\}}_{q,[[s]]_I}$
ϵ	$\{p\}$	$\{a\}$	$\{q, a.\mathbf{0}\}$	$\{a\}$
a	$\{0, b.\mathbf{0}\}$	$-$	$\{0\}$	$-$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

When analysing the sets $(p \text{ after } [[\epsilon]]_{\{a\}}) = \{p\}$ and $(q \text{ after } [[\epsilon]]_{\{a\}}) = \{q, a.\mathbf{0}\}$, we ignore the fact that q starts with a hidden action b; the only relevant residuals are those performing a. With a similar analysis we conclude that the condition on must-sets also holds for set $\{b\}$. Hence, $a.b.\mathbf{0} \approx^{\mathbb{I}}_{\mathsf{ind}} b.a.\mathbf{0}$ holds. □

The following example illustrates also the fact that individualistic observers are unable to track causal dependencies between choices made in different parts of the interface.

Example 4.12. Let $p_1 = a.c.\mathbf{0} + b.d.\mathbf{0}$ and $p_2 = a.d.\mathbf{0} + b.c.\mathbf{0}$ be two alternative implementations for a service with interface $\mathbb{I} = \{\{a, b\}, \{c, d\}\}$. These two processes are distinguished by the uncoordinated preorder $(p_1 \not\approx^{\{\{a,b\},\{c,d\}\}}_{\mathsf{unc}} p_2)$ because of the observers $o_1 = \overline{a}.\mathbf{1} \parallel \overline{c}.\mathbf{1}$ $(p_1 \not\sqsubseteq^{\{\{a,b\},\{c,d\}\}}_{\mathsf{unc}} p_2)$ and $o_2 = \overline{b}.\mathbf{1} \parallel \overline{c}.\mathbf{1}$ $(p_2 \not\sqsubseteq^{\{\{a,b\},\{c,d\}\}}_{\mathsf{unc}} p_1)$.

They are instead equated by the individualistic preorder with respect to $\mathbb{I}$, $p_1 \approx^{\mathbb{I}}_{\mathsf{ind}} p_2$. Indeed, if only the part of the interface $\{a, b\}$ is of interest, we have that p_1 and p_2 are equivalent because they exhibit the same interactions over channels a and b. Similarly, without any a priori knowledge of the choices made for $\{a, b\}$, the behaviour observed over $\{c, d\}$ can be described by the non-deterministic choice $\tau.c.\mathbf{0} + \tau.d.\mathbf{0}$, and hence, p_1 and p_2 are indistinguishable also over $\{c, d\}$.

We use the alternative characterisation to prove our claim. As usual, $p_1 \Downarrow s$ and $p_2 \Downarrow s$ for any s. The tables below show coincidence of the must-sets. We would only like to remark that $ac \in [[a]]_{\{a,b\}}$ and, consequently, p_1 after $[[a]]_{\{a,b\}}$ contains also process 0.

$[[s]]_{\{a,b\}}$	p_1 after $[[s]]_{\{a,b\}}$	$L^{\{a,b\}}_{p_1,[[s]]_I}$	p_2 after $[[s]]_{\{a,b\}}$	$L^{\{a,b\}}_{p_2,[[s]]_I}$
ϵ	p_1	$\{a,b\}$	p_2	$\{a,b\}$
a	$\{c.\mathbf{0},0\}$	–	$\{d.\mathbf{0},0\}$	–
b	$\{d.\mathbf{0},0\}$	–	$\{c.\mathbf{0},0\}$	–
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$[[s]]_{\{c,d\}}$	p_1 after $[[s]]_{\{a,b\}}$	$L^{\{c,d\}}_{p_1,[[s]]_I}$	p_2 after $[[s]]_{\{a,b\}}$	$L^{\{c,d\}}_{p_2,[[s]]_I}$
ϵ	p_1	$\{c,d\}$	p_2	$\{c,d\}$
c	$\{0\}$	–	$\{0\}$	–
d	$\{0\}$	–	$\{0\}$	–
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

□

5. Relation between must, uncoordinated and individualistic preorders

In this section, we formally study the relationships between the classical must preorder and the two preorders we have introduced. We start by showing that a refinement of an interface induces a coarser preorder, e.g., splitting the observation among more uncoordinated observers decreases the discriminating power of the observers. We say that an interface $\mathbb{I}'$ is a *refinement* of another interface $\mathbb{I}$ when the partition $\mathbb{I}'$ is finer than the partition $\mathbb{I}$.

Lemma 5.1. *Let $\mathbb{I}$ be an interface and $\mathbb{I}'$ a refinement of $\mathbb{I}$. Then, $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$ implies $p \sqsubseteq^{\mathbb{I}'}_{\mathsf{unc}} q$.*

Proof. The proof follows by showing that $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$ implies $p \preceq^{\mathbb{I}'}_{\mathsf{unc}} q$. Let D and D' be the dependency relations induced respectively by $\mathbb{I}$ and $\mathbb{I}'$. Since $\mathbb{I}'$ is a refinement of $\mathbb{I}$, $D' \subseteq D$ and therefore $[s]_D \subseteq [s]_{D'}$ for all s. Assume $p \Downarrow [s]_{D'}$ for $s \in \mathsf{Act}^*$. Then,

- $p \Downarrow t$ for all $t \in [s]_{D'}$. Note that $[t]_{D'} = [s]_{D'}$ because $t \in [s]_{D'}$. Consequently, $p \Downarrow [t]_D$ because $[t]_D \subseteq [t]_{D'} = [s]_{D'}$. Since $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, we know that $q \Downarrow [t]_D$, which implies $q \Downarrow t$. Therefore, $q \Downarrow [s]_{D'}$.
- Assume $L \subseteq I'$, $I' \in \mathbb{I}'$ and (p after $[s]_{D'}$) MUST L. Then, (p after t) MUST L for all $t \in [s]_{D'}$. Therefore, (p after $[t]_D$) MUST L because $[t]_D \subseteq [t]_{D'} = [s]_{D'}$. Since $\mathbb{I}'$ is a refinement of $\mathbb{I}$, there is some $I \in \mathbb{I}$ such that $I' \subseteq I$ and $L \subseteq I$ and $I \in \mathbb{I}$. Consequently, $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$ implies (q after $[t]_D$) MUST L and, hence, (q after t) MUST L for all $t \in [s]_{D'}$. Therefore, (q after $[s]_{D'}$) MUST L. □

This result allows us to conclude that the uncoordinated preorder is coarser than the classical must testing preorder. It suffices to note that the preorder associated to the maximal element of the partition lattice, i.e., the trivial partition $\mathbb{I} = \{\mathsf{Act}\}$, corresponds to $\sqsubseteq_{\mathsf{must}}$.

Lemma 5.2. $\sqsubseteq_{\mathsf{must}} = \sqsubseteq^{\{\mathsf{Act}\}}_{\mathsf{unc}}$.

Proof. ($\subseteq$) We show that $p \preceq_{\mathsf{must}} q$ implies $p \preceq^{\{\mathsf{Act}\}}_{\mathsf{unc}} q$ for all p and q. Assume $p \Downarrow [s]_D$ for $s \in \mathsf{Act}^*$. Then,

- $p \Downarrow t$ for all $t \in [s]_D$. Since $p \preceq_{\mathsf{must}} q$, we know that $p \Downarrow t$ implies $q \Downarrow t$ for all $t \in [s]_D$. Consequently, $q \Downarrow [s]_D$.
- Assume (p after $[s]_D$) MUST L for any $L \subseteq \mathsf{Act}$. Then, (p after t) MUST L for all $t \in [s]_D$. Since $p \preceq_{\mathsf{must}} q$, ($q$ after t) MUST L for all $t \in [s]_D$. Hence, (q after $[s]_D$) MUST L.

($\supseteq$) We show that $p \preceq_{\mathsf{unc}}^{\{\mathsf{Act}\}} q$ implies $p \preceq_{\mathsf{must}} q$ for all p and q. First note that the interface $\{\mathsf{Act}\}$ induces a total dependency relation D on Act. This implies $[s]_D = \{s\}$ for all $s \in \mathsf{Act}^*$. Assume $p \Downarrow s$ for $s \in \mathsf{Act}^*$. Then,

- $p \Downarrow [s]_D$. From $p \preceq_{\mathsf{unc}}^{\{\mathsf{Act}\}} q$, we know that $p \Downarrow [s]_D$ implies $q \Downarrow [s]_D$. Consequently, $q \Downarrow s$.
- Assume that $(p \text{ after } [s]_D)$ MUST L holds for a finite $L \subseteq \mathsf{Act}$. Since $p \preceq_{\mathsf{unc}}^{\{\mathsf{Act}\}} q$, $(q \text{ after } [s]_D)$ MUST L. Hence, $(q \text{ after } s)$ MUST L. □

Corollary 5.3. *Let $\mathbb{I}$ be an interface. Then, $p \sqsubseteq_{\mathsf{must}} q$ implies $p \sqsubseteq_{\mathsf{unc}}^{\mathbb{I}} q$.*

The converse of Lemma 5.1 and Corollary 5.3 do not hold. Consider the processes $p = a.b.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$ and $q = b.a.\mathbf{0} + a.\mathbf{0} + b.\mathbf{0}$. It has been shown, in Example 3.16, that we have $p \sqsubseteq_{\mathsf{unc}}^{\{\{a\},\{b\}\}} q$. Nonetheless, it is easy to check that $p \not\sqsubseteq_{\mathsf{must}} q$ (i.e., $p \not\sqsubseteq_{\mathsf{unc}}^{\{\mathsf{Act}\}} q$) by using $o = \overline{b}.(\tau.\mathbf{1} + \overline{a}.\mathbf{0})$ as observer.

We also have that the individualistic preorder is coarser than the uncoordinated one.

Proposition 5.4. *Let $\mathbb{I}$ be an interface. Then, $p \sqsubseteq_{\mathsf{unc}}^{\mathbb{I}} q$ implies $p \sqsubseteq_{\mathsf{ind}}^{\mathbb{I}} q$.*

Proof. Let D be the dependency relation induced by $\mathbb{I}$. We first note that $[t]_D \subseteq [[s]]_I$ for all $t \in [[s]]_I$ and $I \in \mathbb{I}$ because every two strings in the same Mazurkiewicz trace have the same symbols and symbols in the same part of the interface do not commute. Then, assume $p \Downarrow [[s]]_I$ for $s \in \mathsf{Act}^*$. Consequently,

- $p \Downarrow t$ for all $t \in [[s]]_I$. Since $[[t]]_I = [[s]]_I$ and $[t]_D \subseteq [[s]]_I$, $p \Downarrow t'$ for all $t' \in [t]_D$ and $t \in [[s]]_I$. Moreover, $p \preceq_{\mathsf{unc}}^{\mathbb{I}} q$ implies $q \Downarrow t'$ for all $t' \in [t]_D$ and $t \in [[s]]_I$. Consequently, $q \Downarrow [[s]]_I$.
- Assume $(p \text{ after } [[s]]_I)$ MUST $L \cup (\mathsf{Act} \backslash I)$ with $L \subseteq I$. Then, $(p \upharpoonright I \text{ after } t \upharpoonright I)$ MUST L, for all $t \in [[s]]_I$. Then $p \preceq_{\mathsf{unc}}^{\mathbb{I}} q$ implies $(q \upharpoonright I \text{ after } t' \upharpoonright I)$ MUST L for all $t' \in [t]_D$ and $t \in [[s]]_I$. □

The converse does not hold, i.e., $p \sqsubseteq_{\mathsf{ind}}^{\mathbb{I}} q$ does not imply $q \sqsubseteq_{\mathsf{unc}}^{\mathbb{I}} p$. Indeed, we have that $a.b.\mathbf{0} \sqsubseteq_{\mathsf{ind}}^{\{\{a\},\{b\}\}} b.a.\mathbf{0}$ (Example 4.11) but $a.b.\mathbf{0} \not\sqsubseteq_{\mathsf{unc}}^{\{\{a\},\{b\}\}} b.a.\mathbf{0}$ (Example 3.16).

6. Multiparty testing at work

In this section we show how to use the proposed preorders for analysing a larger scenario involving a replicated data store with alternative policies for consistency. In order to support the task of checking relations, we introduce a prototype implementation of the alternative characterisations provided in the paper limited to the fragment of the calculus with finite processes. We resorted to this limitation because the proposed alternative characterisations use quantification over all possible traces of a process. The development of decision procedures for the infinite case (e.g., along the lines of [BF18]) is left to future work.

6.1. Implementation in Prolog.

To provide the Prolog implementation of the new testing preorders, we rely on their alternative characterisations in term of traces. The actual implementation is restricted to the finite fragment of our specification language (Sequential CCS) and is available from `https://github.com/hmelgra/Multiparty-preorders`.

Processes are represented as functional terms built-up from the constants `0` and `1`, the unary operator `~` (output actions), and the binary functions `*` (prefix) and `+` (choice). The operational semantics of finite CCS processes is given by the ternary predicate `red(P,L,Q)`,

which is defined in one-to-one correspondence with the inference rules for finite processes (i.e., those rules that do not involve recursive processes) in Definition 2.1. The corresponding Prolog predicates are the following.

```
red(1, tick, 0).
red(L * P, L, P).
red(P + _, L, P1) :- red(P, L, P1).
red(_ + Q, L, Q1) :- red(Q, L, Q1).
```

Now, by building on the predicate `red(_,_,_)`, we inductively define the ternary (*weak reduction*) relation $P \stackrel{S}{\Longrightarrow} Q$ as the predicate `wred(P, S, Q)` below.

```
1 wred(P, [], P).
2 wred(P, [L|S], Q) :- red(P, L, R), L\=tau, wred(R, S, Q).
3 wred(P, S, Q) :- red(P, tau, R), wred(R, S, Q).
```

The rules above respectively stand for $P \Longrightarrow P$ (line 1); $P \stackrel{LS}{\Longrightarrow} Q$ if $L \neq \tau$ and $P \stackrel{L}{\longrightarrow} R$ and $R \stackrel{S}{\Longrightarrow} Q$ (line 2); and $P \stackrel{S}{\Longrightarrow} Q$ if $P \stackrel{\tau}{\longrightarrow} R$ and $R \stackrel{S}{\Longrightarrow} Q$ (line 3).

Then, the *set of traces from* P, $S = \mathsf{str}(P)$, is defined as follows.

```
1 tr(P, T) :- wred(P, T, _).
2 str(P, S) :- setof(T, tr(P,T), S).
```

The set containing the residuals of a process P after the execution of a sequence of actions T is defined by the following two rules

```
1 after(P, T, []) :- not(wred(P, T, _)), !.
2 after(P, T, Qs) :- setof(Q, wred(P,T,Q), Qs).
```

The first rule states that P after $T = \emptyset$ when P does not have T as one of its traces, while the second one handles the case in which T is a trace of P. The predicate `after(_,_,_)` is implemented with two rules because `setof(Q, wred(P,T,Q), Qs)` fails when the goal `wred(P,T,Q)` does not have any solution.

The predicate P MUST L of Definition 2.4 is inductively implemented by the following rules.

```
1 must([], _).
2 must([P|Ps], L) :-
3     member(A, L), wred(P, [A], _), !, must(Ps, L).
```

Line 1 stands for the base case, i.e., $\emptyset$ MUST L for any L. Differently, Line 2 states that for a non empty set of processes $\{P\} \cup Ps$, it should be the case that there exists some action $A \in L$ such that $P \stackrel{A}{\Longrightarrow}$ and Ps MUST L.

We have now all the ingredients needed for the definition of $\preceq_{\mathsf{must}}$. Our implementation relies on refutation, i.e., we indirectly show $p \preceq_{\mathsf{must}} q$ by falsifying $p \not\preceq_{\mathsf{must}} q$. From Definition 3.9, we deduce that $p \not\preceq^{\mathbb{I}}_{\mathsf{unc}} q$ if there exist $s \in \mathsf{Act}^*$, $I \in \mathbb{I}$, a finite $L \subseteq I$, such that $p \Downarrow [s]_D$, and either

- $q \Uparrow [s]_D$, or
- $(p \text{ after } [s]_D)$ MUST L and $(q \text{ after } [s]_D)$ $\not\text{MUST}$ L.

Since we are considering the finite fragment of the calculus, the convergence predicate $\Downarrow_s$ trivially holds for finite processes. Consequently, for finite processes we have, $p \not\preceq^{\mathbb{I}}_{\mathsf{unc}} q$ if there exist $s \in \mathsf{Act}^*$, $I \in \mathbb{I}$, and a finite $L \subseteq I$, such that $(p \text{ after } [s]_D)$ MUST L and $(q \text{ after } [s]_D)$ MUST L. Moreover, we take advantage of the refutation procedure to obtain witnesses that explain why two particular processes are not in $\preceq_{\mathsf{must}}$ relation. Hence, we implement $\not\preceq_{\mathsf{must}}$ as the quaternary predicate `notleqmust(P,Q,S,L)` meaning that $P \not\preceq_{\mathsf{must}} Q$ because $(P \text{ after } S)$ MUST L but $(Q \text{ after } S)$ MUST L.

```
1 notleqmust(P, Q, S, L):-
2     str(P+Q, Ss), member(S, Ss),
3     after(P, S, Ps), after(Q, S, Qs),
4     n(P+Q, As), subseteq(L, As),
5     must(Ps, L), not(must(Qs, L)).
6
7 leqmust(P,Q) :- not(notleqmust(P,Q,_,_)).
```

Line 2 states that we only consider the set `Ss` of traces that are either traces of `P` or `Q` and disregard any other trace because the residuals for both `P` and `Q` are empty in those cases, and hence uninteresting. When defining must-sets, it is useless to consider actions that are not in the alphabet of the processes[1]. Therefore, line 4 states that we only consider subsets `L` of the names occurring in either `P` or `Q` . Then, in order to show that `P` and `Q` are not in $\preceq_{\mathsf{must}}$ relation, we search for a set `L` that is a must-set of the residuals of `P` after `S` (i.e., `must(Ps, L)`) but not of the residuals of `Q` after `S` (line 5). Finally, the predicate $\preceq_{\mathsf{must}}$ is just defined as the negation of $\not\preceq_{\mathsf{must}}$ (line 7).

As an example of use of the `notleqmust(_,_,_,_)` predicate, we can use it to show that neither $\mathbf{0} \sqsubseteq_{\mathsf{must}} \tau.a.\mathbf{0} + \tau.b.\mathbf{0}$ nor $\tau.a.\mathbf{0} + \tau.b.\mathbf{0} \sqsubseteq_{\mathsf{must}} \mathbf{0}$ hold.

In fact, the following query

```
1 ?- notleqmust(0, tau * a * 0 + tau * b * 0,S,L).
```

has several solutions, among which we have `S = [a], L = []`.

Similarly,

```
1 ?- notleqmust(tau * a * 0 + tau * b * 0, 0, S,L).
```

has `S = [], L = [a, b]` among its solutions.

The implementation for the uncoordinated and individualistic preorders follows analogously. First, we generalise the definition of residuals to consider a set of traces instead of just a trace. This is done just by collecting all the residuals of the process for each trace in the set. We use the ternary predicate `afterC(_,_,_)` defined as follows.

```
1 afterC(_,[],[]).
2 afterC(P,[X|Xs],Ps):- after(P,X,P1s), afterC(P,Xs,P2s),
3     union(P1s,P2s,Ps).
```

In addition, we use two auxiliary predicates: `independence(I,Ind)`, which computes the independence relation `Ind` induced by an interface `I`; and `mazurkiewicz(Ind,S,CT)`,

[1]The definition of predicate `n(P,As)`, which computes the alphabet of `P`, has been omitted because it is straightforward.

which takes an independence relation `Ind`, a set of traces `S` belonging to the same equivalence class, and generates the complete set of traces `CT` in that equivalence class. We omit here their definition because are straightforward and not interesting. We first define the relation `maximalPref(_,_,_,_)` for the set of maximal reductions `MR` of a process `P` within the equivalence class of `T` (for the interface `I`).

```
1 maximalPref(P,T,I,MR) :- independence(I,Ind), mazurkiewicz(Ind,[T],CT),
2     str(P,SP), prefs(SP,CT,PSP), maximal(PSP,PSP,MR),!.
```

We compute the independence relation `Ind` induced by the interface `I` and the Mazurkiewicz class `CT` of `T`. Then, we select the maximal elements `MR` from the set of reductions `SP` of `P` for the class `CT` (the omitted implementations of `prefs(_,_,_)` and `maximal(_,_,_)` are uninteresting). Then, the actual set of strings for Mazurkiewicz trace is the set of maximal prefixes, if they jointly-complete the trace (i.e., the omitted predicate `dagger(_,_,_)`); otherwise is empty.

```
1 strClass(P,T,I,MR):- maximalPref(P,T,I,MR), dagger(MR,T,I).
2 strClass(P,T,I,[]):- maximalPref(P,T,I,MR), not(dagger(MR,T,I)).
```

As for the classical must preorder, we implement $\preceq^{\mathbb{I}}_{\mathsf{unc}}$ in terms of $\not\preceq^{\mathbb{I}}_{\mathsf{unc}}$, which is defined by the predicate `notlequnc(P,Q,I,T,L)`, in which the additional parameter `I` stands for the interface. Its definition is below.

```
1 notlequnc(P,Q,I,T,L):-
2     str(P+Q,Ts), !, member(T,Ts),
3     strClass(P,T,I,CTP), strClass(Q,T,I,CTQ),
4     afterC(P,CTP,Ps), afterC(Q,CTQ,Qs),
5     member(PI, I), subseteq(L, PI),
6     must(Ps, L), not(must(Qs,L)).
7
8 lequnc(P,Q,I) :- not(notlequnc(P,Q,I,_,_)).
```

The differences with respect to the definition of `notlequnc(_,_,_,_)` are the following:

- we compute the set of strings with respect to an equivalence class, i.e., `CTP` and `CTQ` (line 3);
- residuals are obtained for each equivalence class of a trace (line 4) (instead of just a string);
- must-sets are built with actions in just one part of the interface (line 5).

Then, we can check, e.g., that $\tau.a.\mathbf{0} + \tau.b.\mathbf{0} \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} 0$ for $\mathbb{I} = \{\{a\},\{b\}\}$ by executing the query

```
1 ?- notlequnc(tau * a * 0 + tau * b * 0,0,[[a],[b]],T,L).
```

which does not have any solutions.

Also, we can test that $a.b \not\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} b.a$ for $\mathbb{I} = \{\{a\},\{b\}\}$, because the query

```
1 ?- notlequnc(b* a * 0 ,a * b * 0,[[a],[b]], T, L).
```

has several solutions, among which we have `T = [], L = [b]`.

The implementation of the individualistic preorder consists in the definition of an analogous predicate `notleqind(P,Q,I,T,L)`, which considers the equivalence classes of filtered traces instead of the Mazurkiewicz ones. It is defined as follows.

```
1 notleqind(P,Q,I,T,L1):-
2     str(P+Q,Ts), member(T,Ts), member(PI,I),
3     filtered(T,PI,Ts,CT), afterC(P,CT,P1), afterC(Q,CT,Q1),
4     complement(I, PI, C), subseteq(L1, PI), append(L1,C,L),
5     must(P1, L), not(must(Q1,L)).
6
7 leqind(P,Q,I) :- not(notlequnc(P,Q,I,_,_)).
```

In this case the variable `CT` in line 3 stands for the (relevant part of the) equivalence class of the trace `T`. Since the equivalence classes for the filtered case are all infinite and, hence, cannot be computed completely, the predicate `filtered(T,PI,Ts,CT)` simply generates the traces in the equivalence class of `T` that are also traces of at least one of the two processes under comparison (note that the residuals are empty for both processes in the remaining cases, and hence irrelevant). The definition of `filtered(T,PI,Ts,CT)` takes a part of the interface `PI` and a set of traces `Ts`, and returns `CT` which contains the set of traces in `Ts` whose projection over `PI` coincides with the projection of `T`. Note that `Ts` in line 3 corresponds to the traces in either `P` or `Q` (line 2). The remaining difference concerns to the generation of must-sets (line 4). In this case, each candidate must-set `L` contains a subset `L1` of the part of the interface under analysis `PI` and the set `C` containing all actions in the interface `I` that are not in `PI` (this set is computed by the predicate `complement(I, PI, C)`, whose definition has been omitted).

We can use this predicate to check, e.g., that $a.b.\mathbf{0} \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} b.a.\mathbf{0}$ for $\mathbb{I} = \{\{a\},\{b\}\}$ by executing the query

```
1 ?- notleqind( a * b * 0, b * a * 0, [[a],[b]], T,L).
```

which does not have any solution.

Also, we may check that $a.b.\mathbf{0} \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} b.a.\mathbf{0}$ does not hold when $\mathbb{I} = \{\{a,b\}\}$ because the query

```
1 ?- notleqind( a * b * 0, b * a * 0, [[a,b]], T,L).
```

has several solutions, e.g., `T = [], L = [a]`.

We now illustrate the use of the introduced preorders and of our prototype implementation in a larger scenario.

6.2. **A case study.** Distributed, non-relational databases such as Dynamo [DHJ+07] and Cassandra [LM10] provide highly available storage by replicating data and relaxing consistency guarantees. Such databases store key-value pairs that can be accessed by using two operations: `get` to retrieve the value associated with a key, and `put` to store the value of a particular key. A client issuing an operation interacts with the closest server, which plays the role of a coordinator and mediates between the client and the replicas to complete the client request. Each client request is associated with a consistency level, which specifies the degree of consistency required over data. For a `put` operation, the consistency level states the number of replicas that must be written before sending an acknowledgement to the client. Similarly, the consistency level of a `get` operation specifies the number of replicas that must reply to the read request before returning the data to the client. Cassandra provides several

consistency levels; for instance, an operation may request to be performed over just `ONE` or `TWO` replicas, or over the majority of the replicas (i.e., `QUORUM`) or over `ALL` the replicas. Consequently, depending on the consistency level required by the client, the coordinator chooses the replicas to contact.

We will now describe the behaviour of a node acting as coordinator in a configuration that involves two additional replicas. Then we will introduce alternative policies the coordinator might want to use when reacting to users request and will discuss their relationships.

For simplicity reasons, we just illustrate the protocol for processing the operation `get` and abstract away from the values exchanged during the communication (the `put` operation is analogous). The actual protocol for handling a `get` is described below as a `CCS` process.

$$\begin{aligned}
\texttt{Coord} &\stackrel{def}{=} \texttt{get}.(\tau.\overline{\texttt{err}}.\mathbf{0} + \tau.\overline{\texttt{ret}}.\mathbf{0} + \tau.\texttt{Query}_1 \\
&\qquad + \tau.\texttt{Query}_2 + \tau.\texttt{Query}_{1,2} + \tau.\texttt{Query}_{2,1}) \\
\texttt{Query}_i &\stackrel{def}{=} \overline{\texttt{read}_i}.(\tau.\overline{\texttt{err}}.\mathbf{0} + \texttt{ret}_i.(\tau.\overline{\texttt{err}}.\mathbf{0} + \tau.\overline{\texttt{ret}}.\mathbf{0})) \\
\texttt{Query}_{i,j} &\stackrel{def}{=} \overline{\texttt{read}_i}.\overline{\texttt{read}_j}.(\tau.\overline{\texttt{err}}.\mathbf{0} + \texttt{Ans}_{i,j} + \texttt{Ans}_{j,i}) \\
\texttt{Ans}_{i,j} &\stackrel{def}{=} \texttt{ret}_i.(\tau.\overline{\texttt{err}}.\mathbf{0} + \tau.\overline{\texttt{ret}}.\mathbf{0} + \texttt{ret}_j.(\tau.\overline{\texttt{ret}}.\mathbf{0} + \tau.\overline{\texttt{err}}.\mathbf{0}))
\end{aligned}$$

As stated in `Coord`, the coordinator after receiving the request `get` internally decides to either:

- reply to the client with the error message `err`, e.g., when the available nodes are not enough to guarantee the requested consistency level; or
- return the requested information by using just local information (message `ret`); or
- retrieve information by contacting just one of the additional replicas following the protocol defined by $\texttt{Query}_i$; or
- retrieve information from both replicas, following the protocol defined by $\texttt{Query}_{i,j}$.

The protocol followed by the coordinator when contacting replica i is modelled by process $\texttt{Query}_i$: The coordinator sends a read request over the channel $\texttt{read}_i$ and awaits an answer on channel $\texttt{ret}_i$, however it may internally decide not to wait for the answer from the replica and send an error to the client (e.g., in a timeout expires). When the coordinator receives the response from the replica, it may return the requested information to the client or signal an error (e.g., when the consistency level cannot be satisfied by the current state of the replicas).

The protocol followed by the coordinator when contacting both replicas is modelled by process $\texttt{Query}_{i,j}$: When awaiting for their responses, the coordinator may internally decide to reply to the client before or after receiving any of the two answers.

Any equation `name` $\stackrel{def}{=}$ *proc* above can be defined in Prolog by using the predicate `proc(name, proc)` as shown below.

```
proc(coord, get * (tau * ~err * 0 + tau * ~ret * 0
                   + tau * Query1 + tau * Query2
                   + tau * Query12 + tau * Query21))
:- proc(query(1), Query1), proc(query(2), Query2),
   proc(query(1,2), Query12), proc(query(2,1), Query21).


proc(query(I), ~read(I) * (tau * ~err * 0
```

```
                              + ret(I) * (tau * ~err * 0 + tau * ~ret * 0))).

proc(query(I,J), ~read(I) * ~read(J) * (tau * ~err * 0 + AnsIJ + AnsJI))
:- proc(ans(I,J),AnsIJ), proc(ans(J,I),AnsJI).

proc(ans(I,J), ret(I) * (tau * ~ret * 0 + tau * ~err * 0
                    + ret(J) * (tau * ~ret * 0 + tau * ~err * 0))).
```

A possible implementation of $\mathtt{Coord}$ may only provide the part of the protocol that always contact the two additional replicas regardless of the information and consistency level requested by the client. Such implementation can be described as follows,

$$\mathtt{Coord}_1 \stackrel{def}{=} \mathtt{get}.\mathtt{Query}_{1,2}$$

where $\mathtt{Query}_{1,2}$ is as before. This defining equation is implemented in Prolog as follows,

```
proc(coord1, get * Query12) :- proc(query(1,2), Query12).
```

We can check that $\mathtt{Coord} \sqsubseteq_{\mathsf{must}} \mathtt{Coord}_1$ by performing the query

```
?:- proc(coord, Coord), proc(coord1,Coord1), leqmust(Coord,Coord1).
```

An alternative implementation of $\mathtt{Coord}$ may decide to communicate an error to the client but still accept responses from the replicas after this interaction. This feature allows the coordinator to update its local state with information that can be used when answering future requests. Such implementation can be described as follows:

$$\begin{array}{rcl} \mathtt{Coord}_2 & \stackrel{def}{=} & \mathtt{get}.\overline{\mathtt{read}_1}.\overline{\mathtt{read}_2}.(\tau.\overline{\mathtt{err}}.\mathtt{ret}_1.\mathtt{ret}_2.\mathbf{0} + \mathtt{Wait}_{1,2} + \mathtt{Wait}_{2,1}) \\ \mathtt{Wait}_{i,j} & \stackrel{def}{=} & \mathtt{ret}_i.(\tau.\overline{\mathtt{ret}}.\mathtt{ret}_j.\mathbf{0} + \tau.\overline{\mathtt{err}}.\mathtt{ret}_j.\mathbf{0} + \mathtt{ret}_j.(\tau.\overline{\mathtt{ret}}.\mathbf{0} + \tau.\overline{\mathtt{err}}.\mathbf{0})) \end{array}$$

Note that $\mathtt{Coord}_2$ accepts responses from the replicas even after it has replied to the client. As for $\mathtt{Coord}$, the definition of $\mathtt{Coord}_2$ in Prolog is straightforward (and omitted here). When considering the classical must testing preorder, it holds that $\mathtt{Coord} \not\sqsubseteq_{\mathsf{must}} \mathtt{Coord}_2$. However, as far as the behaviour of the client and the replicas is concerned, the implementation of $\mathtt{Coord}_2$ is harmless. In fact, we can prove that $\mathtt{Coord} \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \mathtt{Coord}_2$ when

$$\mathbb{I} = \{\{\mathtt{get}, \mathtt{ret}, \mathtt{err}\}, \{\mathtt{get}, \mathtt{read}_1, \mathtt{ret}_1\}, \{\mathtt{get}, \mathtt{read}_2, \mathtt{ret}_2\}\}$$

For convenience, when querying the program we add the following definition rule for the interface.

```
int([[get,~ret,~err], [~read(1),ret(1)],[~read(2),ret(2)]]).
```

and then query the program as follows:

```
?- proc(coord, C), proc(coord2,C2), int(I), notlequnc(C,C2,I,_,_).
```

The query above has no solutions, and hence $\mathtt{Coord} \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \mathtt{Coord}_2$. Similarly, it can be proved that $\mathtt{Coord}_1 \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \mathtt{Coord}_2$. On the contrary, it can be checked the neither $\mathtt{Coord}_2 \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \mathtt{Coord}$ nor $\mathtt{Coord}_2 \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \mathtt{Coord}_1$. For instance, the query

```
?- proc(coord2,C2), proc(coord1, C1), int(I), notlequnc(C2,C1,I,S,L).
```

has several solutions, e.g.,:

- S = [get, ~read(1), ~read(2), ret(1), ~err], L = [ret(2)],
- S = [get, ~read(1), ~read(2), ret(2), ~err], L = [ret(1)],
- S = [get, ~read(1), ~read(2), ~err], L = [ret(1)],
- S = [get, ~read(1), ~read(2), ~err, ret(1)],L = [ret(2)].

All of them show that $\mathtt{Coord}_2$ is able to accept an answer from a replica even after signaling an error to the client, while $\mathtt{Coord}_1$ is not. Consequently, a replica may distinguish the behaviours of the different implementations: when interacting with $\mathtt{Coord}_1$, a replica i may discover that the coordinator has sent the message $\overline{\mathtt{err}}$ to the client because the interaction $\overline{\mathtt{ret}_i}$ cannot not take place.

We now consider a variant of $\mathtt{Coord}_2$ that chooses a different order for contacting replicas, defined as follows:

$$\mathtt{Coord}_3 \stackrel{def}{=} \mathtt{get}.\overline{\mathtt{read}_2}.\overline{\mathtt{read}_1}.(\tau.\overline{\mathtt{err}}.\mathtt{ret}_1.\mathtt{ret}_2.\mathbf{0} + \mathtt{Wait}_{1,2} + \mathtt{Wait}_{2,1})$$

where $\mathtt{Wait}_{i,j}$ is defined as before. The only difference between $\mathtt{Coord}_2$ and $\mathtt{Coord}_3$ is the order in which $\overline{\mathtt{read}_1}$ and $\overline{\mathtt{read}_2}$ are executed.

We have that $\mathtt{Coord}_2$ and $\mathtt{Coord}_3$ are still distinguishable in the uncoordinated preorder. For instance, the query

```
?- proc(coord2, C2), proc(coord3,C3), int(I), notlequnc(C2,C3,I,S,L).
```

has among it solutions the following one:

```
S = [get], L = [~read(1)]
```

showing that $\mathtt{Coord}_2 \not\sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} \mathtt{Coord}_3$. The test associated with the above witness is built by preventing the interaction with the replica 2 (i.e., when the communication over $\mathtt{read}_2$ is not enabled). However, if the interaction with the replicas is guaranteed, both implementations should be deemed as indistinguishable. In fact, $\mathtt{Coord}_2$ and $\mathtt{Coord}_3$ are indistinguishable in the individualistic preorder. We remark, however, that $\mathtt{Coord}_1$ is still not equivalent to either $\mathtt{Coord}_2$ or $\mathtt{Coord}_3$. For instance, the pair

```
S = [get, ~read(1)], L = [~ret(1)]
```

witnesses the fact that $\mathtt{Coord}_3 \not\sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} \mathtt{Coord}_1$. In fact, while $\mathtt{Coord}_3$ ensures that it will always receive the reply from the replica 1 after sending the request $\mathtt{read}_1$. This is not the case for $\mathtt{Coord}_1$, which may refuse to communicate over $\mathtt{read}_1$, e.g., after an internal timeout.

7. Conclusions and related work

In this paper we have explored different relaxations of the must testing preorder tailored to define new behavioural relations that, in the framework of Service Oriented Computing, are better suited to study compliance between contracts exposed by clients and servers interacting via synchronous binary communication primitives.

In particular, we have considered two different scenarios in which contexts of a service are represented by processes with distributed control. The first variant, that we called uncoordinated preorder, corresponds to multiparty contexts without runtime communication between peers but with the possibility of one peer to block another if it does not perform the expected action. Indeed, the observations at the basis of our experiments are designed with

the assumption that the users of a service interact only via dedicated ports but might be influenced by the fact that other partners do not perform the expected actions. The second preorder we introduced is called individualistic preorder. It accounts for partners that are completely independent from the behaviour of the other ones. Indeed, from a viewpoint of a client, actions by other clients are considered unobservable.

We have shown that the discriminating power of the induced equivalences decreases as observers become weaker; and thus that the individualistic preorder for a given interface is coarser than the uncoordinated preorder for the same interface, which in turn is coarser than the classical testing preorder. As future work we plan to consider different "real life" scenarios and to assess the impact of the different assumptions at the basis of the new preorders and the identifications/orderings they induce. We plan also to perform further studies to get a fuller account, possibly via axiomatisations, of their discriminating power. In the near future, we will also consider the impact of our testing framework on calculi based on asynchronous interactions.

Several variants of the must testing preorder, contract compliance and sub-contract relation have been developed in the literature to deal with different aspects of services compositions, such as buffered asynchronous communication [BZ08, Pad10, MYH09], fairness [Pad11], peer-interaction [BH13]. We have however to remark that these approaches deal with aspects that are somehow orthogonal to the discriminating power of the distributed tests analysed in this work. Our preorders have some similarities with those relying on buffered communications in that both aim at guaranteeing that actions performed by independent peers can be reordered, but we rely on synchronous communication and, hence, message reordering is not obtained by swapping buffered messages but by relying on more generous observers. As mentioned above, we have left the study of distributed tests under asynchronous communication as a future work. However, we would like to remark that, even the uncoordinated and the individualistic preorders are different from those in [BZ08, Pad10, MYH09] that permit explicit action reordering. The paradigmatic example is the equivalence $a.c + b.d \approx_{\mathsf{ind}}^{\{a,b\},\{c,d\}} a.d + b.c$, which does not hold for any of the preorders with buffered communication. The main reason is that, even in presence of buffered communication, the causality, e.g., between a and c is always observed.

Acknowledgments

We are very grateful to Marzia Buscemi for interesting discussions during early stages of this work. We thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions. The work has been partially supported by CONICET (under project PIP 11220130100148CO), UBA (under project UBACYT 20020130200092BA), and MIUR under PRIN projects CINA and IT-Matters.

References

[BBO12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 191–202. ACM, 2012. doi:10.1145/2103656.2103680.

[BF18] Giovanni Tito Bernardi and Adrian Francalanza. Full-abstraction for client testing preorders. *Sci. Comput. Program.*, 168:94–117, 2018. doi:10.1016/j.scico.2018.08.004.

[BH13] Giovanni Bernardi and Matthew Hennessy. Mutually testing processes - (extended abstract). In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2013. doi:10.1007/978-3-642-40184-8_6.

[BZ08] Mario Bravetti and Gianluigi Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundam. Informaticae*, 89(4):451–478, 2008. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi89-4-05.

[BZ09] Mario Bravetti and Gianluigi Zavattaro. Contract-based discovery and composition of web services. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*, volume 5569 of *Lecture Notes in Computer Science*, pages 261–295. Springer, 2009. doi:10.1007/978-3-642-01918-0_7.

[CGP08] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 261–272. ACM, 2008. doi:10.1145/1328438.1328471.

[CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5):19:1–19:61, 2009. doi:10.1145/1538917.1538920.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007. doi:10.1145/1294261.1294281.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010. doi:10.1145/1773912.1773922.

[LP07] Cosimo Laneve and Luca Padovani. The *Must* preorder revisited. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007. doi:10.1007/978-3-540-74407-8_15.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986. doi:10.1007/3-540-17906-2_30.

[Maz95] Antoni W. Mazurkiewicz. Introduction to trace theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995. doi:10.1142/9789814261456_0001.

[Mil89] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[MYH09] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009. doi:10.1007/978-3-642-00590-9_23.

[NH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984. doi:10.1016/0304-3975(84)90113-0.

[NM15] Rocco De Nicola and Hernán C. Melgratti. Multiparty testing preorders. In Pierre Ganty and Michele Loreti, editors, *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*, volume 9533 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2015. doi:10.1007/978-3-319-28766-9_2.

[Pad10] Luca Padovani. Contract-based discovery of web services modulo simple orchestrators. *Theor. Comput. Sci.*, 411(37):3328–3347, 2010. doi:10.1016/j.tcs.2010.05.002.

[Pad11] Luca Padovani. Fair subtyping for multi-party session types. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2011. doi:10.1007/978-3-642-21464-6_9.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.

Appendix A. Proof Details of results in Section 4

In this section we provide the detailed proofs of results in Section 4.

Lemma 4.7. *If $p \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} q$ then for all $s \in \mathsf{Act}^*$ and $I \in \mathbb{I}$, we have that $p \Downarrow [[s]]_I$ implies*

(1) *$q \Downarrow [[s]]_I$*
(2) *$s \in \mathsf{str}(q)$ implies that there exists $t \in [[s]]_I$ such that $t \in \mathsf{str}(p)$.*

Proof.

(1) By contradiction. Suppose there exists $s = a_1 \dots a_n$ such that $p \Downarrow [[s]]_I$ and $q \Uparrow [[s]]_I$. Then, take the observer o_i defined as follows

$$oi = \tau.\mathbf{1} + \overline{b^i_1}.(\tau.\mathbf{1} + \dots (\tau.\mathbf{1} + \overline{b^i_{k_i}}.\tau.\mathbf{1}) \dots)$$

with $s_i = s \upharpoonright I = b^i_1 \dots b^i_{k_i}$. Then, $p \upharpoonright I$ must o_i and $q \upharpoonright I$ $\not\mathsf{must}$ o_i.

Note that $o_i \Downarrow$. Since $p \Downarrow [[s_i]]_I$, every maximal computation of $(p \upharpoonright I) \| o_i$ does not diverge. For each maximal computation $(p \upharpoonright I) \| o_i \Longrightarrow (p' \upharpoonright I) \| o'_i \not\rightarrow$ we proceed by unzipping the computation to conclude that $o_i \stackrel{t}{\Longrightarrow} o'_i \not\rightarrow$ for some t. It can be shown by straightforward induction on the length of the reduction that $o_i \stackrel{t}{\Longrightarrow} o'_i \not\rightarrow$ implies $o_i \stackrel{t}{\Longrightarrow} \mathbf{1} \stackrel{\checkmark}{\longrightarrow} \mathbf{0} = o'_i$. Consequently, each maximal computation of $(p \upharpoonright I) \| o_i$ is successful and p must o_i.

Since $q \Uparrow [[s_i]]_I$ there exists $t \in [[s_i]]_I$ such that $t = t_1 t_2$ and $q \stackrel{t_1}{\Longrightarrow} q'$ and $q' \Uparrow$. As before, we can conclude that $o_i \stackrel{\overline{t_1} \upharpoonright I}{\Longrightarrow} o'_i \stackrel{\overline{t_2} \upharpoonright I}{\Longrightarrow}$. It can be shown by induction on the length of $\overline{t_1}$ that there exists an unsuccessful computation $o_i \stackrel{\overline{t_1} \upharpoonright I}{\Longrightarrow} o'_i$ such that either $o''_i = \tau.\mathbf{1}$ or $o''_i = (\tau.\mathbf{1} + a. \dots)$. Then, there exists a maximal (divergent) unsuccessful computation $(q \upharpoonright I) \| o_i \Longrightarrow (q' \upharpoonright I) \| o''_i \rightarrow (q'' \upharpoonright I) \| o''_i \rightarrow \dots$. Consequently, q $\not\mathsf{must}$ o and this contradicts the assumption $p \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} q$.

(2) By contradiction. Suppose there exists $s = a_1 \dots a_n$ such that $p \Downarrow [[s]]_I$, $s \in \mathsf{str}(q)$ and for all $t \in [[s]]_I$, $t \notin \mathsf{str}(p)$. Then, choose o_i defined as follows

$$o_i = \tau.\mathbf{1} + \overline{b^i_1}.(\tau.\mathbf{1} + \dots (\tau.\mathbf{1} + \overline{b^i_{k_i}}.\mathbf{0}) \dots)$$

with $s_i = s \upharpoonright I = b^i_1 \dots b^i_{k_i}$.

Note that $(q \upharpoonright I)$ $\mathsf{mu\not st}$ o_i because there is a maximal unsuccessful computation of $(q \upharpoonright I) \| o_i$. Since $t \in [[s]]_I$, $t \upharpoonright I = t' b^i_{k_i}$. Without lost of generality, we assume $t = t'' b^i_{k_i}$ $t \notin \mathsf{str}(p)$. Then, either (i) $(p \upharpoonright I) \stackrel{t'}{\Longrightarrow} \stackrel{b^i_{k_i}}{\not\longrightarrow}$ or (ii) $(p \upharpoonright I) \stackrel{t'}{\not\Longrightarrow}$. Case (i) follows immediately, because $o_i \stackrel{t'}{\Longrightarrow} o'_i \not\rightarrow$ implies $o_i \stackrel{t'}{\Longrightarrow} \mathbf{1} \stackrel{\checkmark}{\longrightarrow} \mathbf{0} = o'$. Hence, $(p \upharpoonright I)\mathsf{must}\ o_i$, which is in contradiction with the assumption that $p \sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} q$. Case (ii) follows by noting that $(p \upharpoonright I) \stackrel{t'}{\not\Longrightarrow}$ implies that there exists t_1, t_2 with $t_2 \neq \epsilon$ such that $p \stackrel{t_1}{\Longrightarrow} \stackrel{t_2}{\not\Longrightarrow}$. Moreover, if $o_i \stackrel{t_1}{\Longrightarrow} o'_i \not\rightarrow$ implies $o_i \stackrel{t_2}{\Longrightarrow} \mathbf{1} \stackrel{\checkmark}{\longrightarrow} \mathbf{0} = o'_i$. Consequently, every maximal computation of $(p \upharpoonright I) \| o_i$ is successful and $(p \upharpoonright I)$ $\mathsf{mu\not st}$ o_i, which is in contradiction with the assumption $p \sqsubseteq^{\mathbb{I}}_{\mathsf{unc}} q$. □

Lemma 4.8. *if* $(p$ after $[[s]]_I)$ $\mathsf{MU\not ST}$ L *for some* $L \subseteq \mathsf{Act}$*, then* $\exists t \in [[s]]_I : t \in \mathsf{str}(p)$.

Proof. Suppose $\forall t \in [[s]]_I$, $t \notin \mathsf{str}(p)$. Then $(p$ after $[[s]]_I) = \emptyset$ and, by definition, $\emptyset$ MUST L for every finite $L \subseteq \mathsf{Act}$. □

Lemma 4.9. *If* $p \preceq^{\mathbb{I}}_{\mathsf{ind}} q$*,* $s \in \mathsf{str}(q)$ *and* $p \Downarrow [[s]]_I$ *with* $I \in \mathbb{I}$ *then* $t \in ([[s]]_I \cap \mathsf{str}(p))$.

Proof. Assume $\forall t \in [[s]]_I : t \notin \mathsf{str}(p)$. By Lemma 4.8, $(p$ after $[[s]]_I)$ MUST L for every finite $L \subseteq \mathsf{Act}$. By straightforward induction on the length of the reduction, we can show that $p \stackrel{t}{\Longrightarrow} q$ implies $\mathtt{n}(q) \subseteq \mathtt{n}(p)$. Analogously, we can show that $\mathsf{init}(q) \subseteq \mathtt{n}(p)$. Consequently, $\bigcup\{\mathsf{init}(q') \mid q \stackrel{[[s]]_I}{\Longrightarrow} q'\} \subseteq \mathtt{n}(p)$. Since, $\mathtt{n}(p)$ is finite, we can conclude that the set $\bigcup\{\mathsf{init}(q') \mid q \stackrel{[[s]]_I}{\Longrightarrow} q'\}$ is finite. Therefore, we can find an action a such that for all $t \in [[s]]_I$ we have $q \stackrel{ta}{\not\Longrightarrow}$. Then $(q$ after $[[s]]_I)$ $\mathsf{MU\not ST}$ $\{a\}$ while $(p$ after $[[s]]_I)$ MUST $\{a\}$, which contradicts the hypothesis $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$. □

Theorem 4.10. $\sqsubseteq^{\mathbb{I}}_{\mathsf{ind}} = \preceq^{\mathbb{I}}_{\mathsf{ind}}$.

Proof.

(⊆) Actually we prove that $p \not\preceq^I_{\mathsf{ind}} q$ implies $p \not\sqsubseteq^I_{\mathsf{ind}} q$. Assume that there exists $s = a_1 \dots a_n$ and $L \subseteq I$ such that
(1) $p \Downarrow [[s]]_I$ and $q \Uparrow [[s]]_I$, or
(2) $s \in \mathsf{str}(q)$ and $\forall t \in [[s]]_I . t \notin \mathsf{str}(p)$ or
(3) $(p$ after $[[s]]_I)$ MUST $L \cup (\mathsf{Act} \backslash I)$ and $(q$ after $[[s]]_I)$ $\mathsf{MU\not ST}$ $L \cup (\mathsf{Act} \backslash I)$
For each case we show that there exists an observer such that $p \upharpoonright I$ must o and $q \upharpoonright I$ $\mathsf{mu\not st}$ o. For the two first cases, we take the observers as defined in proof of Lemma 4.7. For the third one, define

$$o = \tau.\mathbf{1} + \overline{b^i_1}.(\tau.\mathbf{1} + \dots (\tau.\mathbf{1} + \overline{b^i_{k_i}}. \sum_{a \in L} a.\mathbf{1}) \dots)$$

with $s \upharpoonright I = b^i_1 \dots b^i_{k_i}$.

(⊇) We prove $p \preceq^I_{\mathsf{ind}} q$ implies $p \sqsubseteq^I_{\mathsf{ind}} q$. Actually, the proof follows by showing that $p \preceq^I_{\mathsf{ind}} q$ and $q \upharpoonright I$ $\mathsf{mu\not st}$ o imply $p \upharpoonright I$ $\mathsf{mu\not st}$ o. Assume there exists an unsuccessful computation

$$q \upharpoonright I \parallel o = q_0 \upharpoonright I \parallel o_0 \stackrel{\tau}{\rightarrow} \dots \stackrel{\tau}{\rightarrow} q_k \upharpoonright I \parallel o_k \stackrel{\tau}{\rightarrow} \dots$$

Consider the following cases:

(1) **The computation is finite**, i.e., there exists n such that $q_n \upharpoonright I \parallel o_n \overset{\mathcal{I}}{\not\rightarrow}$ and $q_i \upharpoonright I \Downarrow$ and $o_i \Downarrow$ for $i \leq n$. By unzipping the computation, there exists s such that $q_0 \upharpoonright I \overset{s}{\Longrightarrow} q_n \upharpoonright I$ and $o_0 \overset{\overline{s}}{\Longrightarrow} o_n$ unsuccessful. Note that $\mathtt{n}(s) \subseteq I$ and hence $s \upharpoonright I = s$ Moreover, $q_n \upharpoonright I$ MUST $\mathsf{init}(o_n)$ and, hence $(q \upharpoonright I \ \mathsf{after}\ s)$ MUST $\mathsf{init}(o_n)$. By Lemma 4.5 (4), we have that $(q \ \mathsf{after}\ [[s]]_I)$ MUST $\mathsf{init}(o_n)$.
   (a) Case $p \Uparrow [[s]]_I$. By Lemma 4.5 (3), $p \upharpoonright I \Uparrow s$. Consequently, there is an unsuccessful computation of $p \upharpoonright I \parallel o$.
   (b) Case $p \Downarrow [[s]]_I$. Note that $s \in \mathsf{str}(q \upharpoonright I)$. By Lemma 4.7 (2), therefore, $\exists t \in [[s]]_I$ and $t \in \mathsf{str}(p)$. Hence, $(p \ \mathsf{after}\ [[s]]_I) \neq \emptyset$. Moreover, from $p \preceq^I_{\mathsf{ind}} q$ we conclude that $(q \ \mathsf{after}\ [[s]]_I)$ MUST $\mathsf{init}(o)$ implies $(p \ \mathsf{after}\ [[s]]_I)$ MUST $\mathsf{init}(o_n)$. Therefore, exists some $p' \in (p \ \mathsf{after}\ [[s]]_I)$ such that p' MUST $\mathsf{init}(o_n)$, and $p \upharpoonright I \overset{s}{\Longrightarrow} p' \upharpoonright I$ is unsuccessful. Hence, there is an unsuccessful computation of $p \upharpoonright I \parallel o$.

(2) **The computation is infinite**. We consider two cases:
   (a) There exists $s \in \mathsf{str}(q \upharpoonright I)$ and $\overline{s} \in \mathsf{str}(o)$ such that $q \upharpoonright I \Uparrow s$ or $o \Uparrow \overline{s}$. Note that $\mathtt{n}(s) \subseteq I$ and hence $s \upharpoonright I = s$. We proceed by case analysis.
      – $q \upharpoonright I \Uparrow s$: By Lemma 4.5 (3), $q \Uparrow [[s]]_I$. Therefore $p \Uparrow [[s]]_I$ because $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$. By Lemma 4.5 (3), $p \upharpoonright I \Uparrow s$. Therefore, there is an unsuccessful computation of $p \upharpoonright I \parallel o$.
      – $q \upharpoonright I \Downarrow s$ (and $o \Uparrow \overline{s}$): Therefore $q \Downarrow [[s]]_I$ by Lemma 4.5 (3). Therefore $p \Downarrow [[s]]_I$ because $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$. By Lemma 4.9, $\exists t \in [[s]]_I : t \in \mathsf{str}(p)$, hence $p \overset{t}{\Longrightarrow} p'$. By Lemma 4.5 (1) $p \upharpoonright I \overset{t\upharpoonright I}{\Longrightarrow} p' \upharpoonright I$. Note that $t \upharpoonright I = s$. Then, $p \upharpoonright I \overset{s}{\Longrightarrow} p' \upharpoonright I$. Hence, the computation obtained by zipping $p \upharpoonright I \overset{s}{\Longrightarrow} p' \upharpoonright I$ and $o \overset{\overline{s}}{\Longrightarrow} o'$ is unsuccessful.
   (b) $\forall n.(q_n \upharpoonright I) \Downarrow$ and $o_n \Downarrow$. Take $s \in \mathsf{Act}^*$ such that $q \overset{s}{\Longrightarrow} q_n$ and $q \upharpoonright I \Downarrow s$ (this is possible because $q \upharpoonright I \parallel o \Longrightarrow q_n \upharpoonright I \parallel o_n$ is unsuccessful and $\forall i \leq n.(q_i \upharpoonright I) \Downarrow$).
   By Lemma 4.5 (1), $(q \upharpoonright I) \Downarrow s$ implies $q \upharpoonright I \overset{s\upharpoonright I}{\Longrightarrow} q' \upharpoonright I$.
   By Lemma 3.9, $q \overset{s}{\Longrightarrow} q_n$ and $q \Downarrow s$ implies either (i) $p \Uparrow [[s]]_I$ or (ii) $p \Downarrow [[s]]_I$ and $p \overset{[[s]]_I}{\Longrightarrow}$.
      (i) $p \Uparrow [[s]]_I$: $\exists t \in [[s]]_I$ such that $p \Uparrow t$. By Corollary 3.7 (4), $o \overset{\overline{s}}{\Longrightarrow}$ unsuccessful implies $o \overset{\overline{t}}{\Longrightarrow}$ unsuccessful, and hence there is an unsuccessful computation of $p \parallel o$.
      (ii) $p \Downarrow [[s]]_I$ and $p \overset{[[s]]_I}{\Longrightarrow}$. Since $q_n \Downarrow$, in the computation there exists q_m such that $q_n \Longrightarrow q_m \not\rightarrow$ and $q_m \overset{a}{\rightarrow} q_{m+1}$. Moreover, $o_n \Longrightarrow o_m \overset{\overline{a}}{\rightarrow} o_{m+1}$. Consequently, for all L such that $a \notin L$, we have $(q \ \mathsf{after}\ [[s]]_I)$ MUST L. Since $p \preceq^{\mathbb{I}}_{\mathsf{unc}} q$, then for all L such that $a \notin L$, we have $(p \ \mathsf{after}\ [[s]]_I)$ MUST L. Therefore, there exists $p_n \in (p \ \mathsf{after}\ [[s]]_I)$ and $p_n \overset{a}{\Longrightarrow} p_{m+1}$. Hence, for all n there is an unsuccessful computation $p \parallel o \Longrightarrow p_n \parallel o_n \Longrightarrow p_{m+1} \parallel o_{m+1}$. □