Testing for Coordination Fidelity*

Yehia Abd Alrahman[1], Claudio Antares Mezzina[2], and Hugo Torres Vieira[3]

[1] University of Gothenburg | Chalmers University of Technology , Sweden
[2] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Italy
[3] IMT School for Advanced Studies Lucca, Lucca, Italy

Abstract. Operation control in modern distributed systems must rely on decentralised coordination among system participants. In particular when the operation control involves critical infrastructures such as power grids, it is vital to ensure correctness properties of such coordination mechanisms. In this paper, we present a verification technique that addresses coordination protocols for power grid operation control. Given a global protocol specification, we show how we can rely on testing semantics for the purpose of ensuring protocol fidelity, i.e., to certify that the interaction among the grid nodes follows the protocol specification.

1 Introduction

Power generation and distribution have been undergoing a revolution in the past years, on the one hand due to the introduction of different solutions for generation, on the other hand because of the impact that such solutions have on distribution grids. More concretely, having a unique power supplier in a grid is an outdated configuration, instead now the scenario of interest involves multiple power supplies and distribution to potentially all nodes of the grid, in particular when renewable energy sources come into play. The real time requirements of such systems demand automatic mechanisms for operation control, that must be certifiably reliable given their critical nature.

Clearly, the outdated centralised control models of power grids cannot scale with the complexity and heterogeneity of emerging configurations. Instead, decentralised operation control must rely on the *coordination* of distributed remote collaborating parties, for example for the purpose of balancing supply-and-demand. It is however vital that such coordination mechanisms are encompassed with techniques that allow to ensure reliability in a rigorous way.

Several proposals of formal models for distributed coordination can be found in the existing literature, for instance based on tuple spaces (e.g., [9]). We may also find recent proposals for new paradigms for interaction models, such as attribute-based communication [1]. The common goal is to support system modelling in a precise way and exclude unexpected behaviours. Building on such

* Yehia Abd Alrahman is funded by the ERC consolidator grant D-SynMA under the European Union's Horizon 2020 research and innovation programme (grant agreement No 772459).

characterisation of system behaviour, one may then ensure reliability properties by means of verification techniques. We distinguish here the approaches based on behavioural type specifications (cf. [15]) that allow to certify protocol fidelity, i.e., that ensure that interacting parties follow a prescribed protocol of interaction.

In this paper, we build on previous work that introduces a model of coordination protocols for power grid operation control [2]. The key principles underlying the proposal are a global programming model, that allows to reason on grid behaviour as a whole, and a notion of operation control transference via interaction. Intuitively, idle nodes react to synchronisations so as to carry out their part in the operation control, hence interactions authorise nodes much like interactions in a token ring protocol authorise nodes to access the shared resource. Together with a notion of network configuration, in particular network topology and state, the principles above are embedded in a protocol language that is role-agnostic, i.e., that does not specify a priori the parties involved in the communications. This means that the development of coordination protocols may consider generic networks, as expectable, and the concrete association with the network nodes involved in the communications is carried out on-the-fly at runtime based on the transference of the operation control.

Likewise to the (global) protocol specification, also the controllers running in network nodes should be developed targeting a generic setting. The key issue, for the purpose of ensuring that the node controllers interact among them according to the protocol specification, is to certify that the implementation of a node controller exhibits the (local) actions expected at network level. Hence, a notion of observational reasoning over implementations is required in order to certify that an implementation complies with a prescribed protocol of interaction. Ideally, such observational power should be as flexible as possible so to allow for the greatest number of implementations to be deemed compliant with the specification. We therefore rely on *testing* [10] since we may streamline the verification by deriving the testers from the specification and abstract away from implementation details that are not pertinent for the protocol validation.

The global protocol language, presented in the next section, is equipped with an operational semantics that provides the reference model of interaction. We refer the interested reader to previous work [2] for a more detailed presentation of the model, but nevertheless the presentation in this paper is self-contained. To that end we illustrate here the flavour of the language in Section 3 by modelling a protocol that addresses the reconfiguration of a power grid which is new to this paper. In Section 4 we present our novel technical development, starting by the definition of the language model for testers and a technique that may be used to synthesise testers from global protocol specifications. We then show how we can rely on a testing semantics to ensure that an implementation complies with a protocol (Definition 1). Based on this notion of compliance, we may then ensure protocol fidelity (Theorem 1), which attests that implementations follow the prescribed protocol of interaction. Finally, Section 5 includes some concluding remarks.

Table 1: Global Language Syntax

(Protocol)	P	$::=$	$\mathbf{0}$
		$\mid$	$\mathrm{P} \mid \mathrm{Q}$
		$\mid$	$\mathbf{rec}\ X.\mathrm{P}$
		$\mid$	X
		$\mid$	S
		$\mid$	$(id)\mathrm{P}$
(Summation)	S	$::=$	$[f^d]_i^o\mathrm{P}$
		$\mid$	$\mathrm{S} + \mathrm{S}$
(Direction)	d	$\in$	$\{\star, \blacktriangle, \blacktriangleright, \bullet\}$

2 A Model for Operation Control Protocols

In this section, we present a global language to model operation control protocols governing power networks. Interaction in our language is driven by the structure of the power network, in particular considering radial power networks, i.e., tree-structures where the root provides power to respective subtree. The interaction model also accounts for a notion of proximity so as to capture backup links. Thus, we consider that nodes can interact if they are in a provide/receive power relation or in a neighbouring relation. In order to identify the target of a synchronisation, specifications include a direction that determines the type of the relation.

As anticipated in the Introduction, the language model embeds the principle that control is transferred by means of synchronisations. For example, a node enabling synchronisation on an action and another one reacting to it may synchronise and the enabling node yields the control to the reacting one. Consequently, the reacting node may enable the next step of the protocol. We may therefore consider that *active* nodes enable synchronisations and, as a consequence of a synchronisation, transfer the *active* role to the reacting node. This allows to specify protocols as a (structured) set of interactions without prescribing the actual identities of the nodes involved, as these are determined operationally due to the transference of the active role in synchronisations.

The syntax of the language is given in Table 1. We use id to range over node identifiers, f to range over synchronisation action labels, and $c, i, o, \ldots$ to range over logical conditions. The protocols, ranged over by P and Q, consist of static specifications and the *active* node construct $(id)\mathrm{P}$, which says that the node with identifier id is active to carry out the protocol P. Static specifications represent the behaviour of the protocol which is defined by termination $\mathbf{0}$, fork $\mathrm{P} \mid \mathrm{Q}$ to specify that both P and Q are to be carried out, and infinite behaviour defined in terms of the recursion $\mathbf{rec}\ X.\mathrm{P}$ and recursion variable X, with the usual meaning. Static specifications also include synchronisation summations $(\mathrm{S}, \ldots)$, where

$S_1 + S_2$ states that either S_1 or S_2 is to be carried out (exclusively), and where $[f^d]_i^o$ represents a synchronisation action: a node active on $[f^d]_i^o P$ that satisfies condition o may synchronise on f with the node(s) identified by the direction d for which condition i holds, leading to the activation of the latter node(s) on protocol P. Intuitively, the node active on $[f^d]_i^o P$ enables the synchronisation, which results in the reaction of the targeted nodes that are activated to carry out the continuation protocol P.

A direction d specifies the target(s) of a synchronisation; $\star$ targets all (children) of the enabling node; $\blacktriangle$ targets the parent; $\blacktriangleright$ targets a neighbour; and $\bullet$ targets the enabler itself and is used to capture local computation steps. We remark that since one node can supply power to several others, synchronisations with $\star$ direction may actually involve several reacting nodes. Any $\star$ synchronisations are therefore interpreted as broadcasts, i.e., $\star$ targets all (direct) children that satisfy the reacting condition, which can be the empty set (e.g., the node has no children or none of them satisfy the reacting condition). Binary interaction, on the other hand, is interpreted as synchronisation and can only occur if the identified target node satisfies the reacting condition.

Example 1. Consider the protocol: $(id)([f_1{}^\star]_{i_1}^{o_1} P_1 + [f_2{}^\bullet]_{i_2}^{o_2}(P_2 \mid P_3))$ which specifies that node id is active to synchronise on f_1 or f_2, exclusively.

There are two language mechanisms, illustrated in Example 1, that may introduce concurrency in the model. One is the broadcast which may lead to the activation of several nodes: in the example, each one of the nodes reacting to f_1 will carry out P_1. Another is the fork construct which allows a node to concurrently carry out two subprotocols: in the example, a node active on $(P_2 \mid P_3)$ will carry out both P_2 and P_3, and potentially synchronise with different nodes in each one.

The semantics of the language relies on a structural congruence relation $\equiv$. Structural congruence is the least congruence relation on protocols that satisfies the rules given in Table 2. The first set of rules states that fork and summation are associative and commutative, and that fork has identity element $\mathbf{0}$ (notice that $\mathbf{0}$ is syntactically excluded from summations). Rule $(id)(P \mid Q) \equiv (id)P \mid (id)Q$ states that a node id is active on both branches of the fork construct. Rule $(id_1)(id_2)P \equiv (id_2)(id_1)P$ states that the order of active nodes is immaterial and rule $(id)\mathbf{0} \equiv \mathbf{0}$ states that a node active on $\mathbf{0}$ is equivalent to $\mathbf{0}$. Intuitively, structural congruence rewriting allows active nodes to "float" in the term towards the synchronisation actions. The rule for recursion unfolding is standard.

Example 2. Considering the active node distribution in a fork, we have that $[f_1{}^\star]_{i_1}^{o_1} P_1 + [f_2{}^\bullet]_{i_2}^{o_2}(id)(P_2 \mid P_3) \equiv [f_1{}^\star]_{i_1}^{o_1} P_1 + [f_2{}^\bullet]_{i_2}^{o_2}((id)P_2 \mid (id)P_3)$

The definition of the semantics depends on the state of the network and on the fact that nodes satisfy certain logical conditions. We consider state information for each node so as to capture both "local" information about the topology (such as the identities of the power provider and of the set of neighbours) and other information relevant for condition assessment (such as the status of the power supply). The network state, denoted by Δ, is a mapping from node identifiers to states, where a state, denoted by s, is a register $id[id', t, n, k, a, e, g]$ containing the

Table 2: Structural Congruence

$\mathrm{P} \,\vert\, \mathbf{0} \equiv \mathrm{P}$	$\mathrm{P}_1 \,\vert\, (\mathrm{P}_2 \,\vert\, \mathrm{P}_3) \equiv (\mathrm{P}_1 \,\vert\, \mathrm{P}_2) \,\vert\, \mathrm{P}_3$	$\mathrm{P}_1 \,\vert\, \mathrm{P}_2 \equiv \mathrm{P}_2 \,\vert\, \mathrm{P}_1$
$\mathbf{rec}\ X.\mathrm{P} \equiv \mathrm{P}[\mathbf{rec}\ X.\mathrm{P}/X]$	$\mathrm{S}_1 + (\mathrm{S}_2 + \mathrm{S}_3) \equiv (\mathrm{S}_1 + \mathrm{S}_2) + \mathrm{S}_3$	$\mathrm{S}_1 + \mathrm{S}_2 \equiv \mathrm{S}_2 + \mathrm{S}_1$
$(id)(\mathrm{P} \,\vert\, \mathrm{Q}) \equiv (id)\mathrm{P} \,\vert\, (id)\mathrm{Q}$	$(id_1)(id_2)\mathrm{P} \equiv (id_2)(id_1)\mathrm{P}$	$(id)\mathbf{0} \equiv \mathbf{0}$

following information: id is the node identifier; id' identifies the power provider; t captures the status of the input power connection; n is the set of identifiers of neighbouring nodes; k is the power supply capacity of the node; a is the number of power supply links (i.e., the number of nodes that receive power from this one); g is the identity of a nearby power generator; and e is the number of power supply links that are in a faulty state. We consider that the elements in the register are natural numbers (and a set of natural numbers for the neighbours) albeit in the examples we use some special symbols (e.g., ∞).

We check conditions against states for the purpose of allowing synchronisations. Given a state s we denote by $s \models c$ that state s satisfies condition c, where we leave the underlying logic unspecified. For example, we may say that $s \models (k > 0)$ to check that s has capacity greater than 0. We also consider a notion of side-effects, in the sense that synchronisation actions may result in state changes so as to model system evolution. By $\mathtt{upd}(id, id', f^d, \Delta)$ we denote the operation that yields the network state obtained by updating Δ considering node id synchronises on f with id', hence the update regards the side-effects of f in the involved nodes. Namely, given $\Delta = (\Delta', id \mapsto s, id' \mapsto s')$ we have that $\mathtt{upd}(id, id', f^d, \Delta)$ is defined as $(\Delta', id \mapsto f^d!(s, id'), id' \mapsto f^d?(s', id))$, where $f^d!(s, id')$ modifies state s according to the side-effects of enabling f^d and considering id' is the reactive node (likewise for the reacting update, distinguished by ?). We consider side-effects only for binary synchronisations (▶ and ▲ directions), where both interacting parties are known, but state changes could also be considered for other directions following similar lines.

The definition of the semantics relies on an auxiliary operation, denoted $d(\Delta, id)$, that yields the recipient(s) of a synchronisation action, given the direction d, the network state Δ and the enabler of the action id. The operation yields the power provider of a node in case the direction is ▲, (any) one of the neighbours in case the direction is ▶, all the nodes that have as parent the enabler in case the direction is $\star$, and is undefined for direction $\bullet$.

The operational semantics is given in terms of configurations consisting of a protocol P and a network state Δ. We use $\Delta; \mathrm{P} \to \Delta'; \mathrm{P}'$ to represent that configuration $\Delta; \mathrm{P}$ evolves in one step to configuration $\Delta'; \mathrm{P}'$, potentially involving state changes (Δ and Δ' may differ) and (necessarily) involving a step in the protocol from P to P$'$.

The semantics of our language is reported in Table 3. Rule Bin captures binary interaction where the direction (d) of the synchronisation action targets either the parent (▲) or a neighbour (▶). Protocol $(id)([f^d]^o_i\mathrm{P} + \mathrm{S})$ states that node id can enable a synchronisation on f provided that the state of id satisfies condition

Table 3: Reduction Rules

$$\frac{d \in \{\blacktriangle, \blacktriangleright\} \quad \Delta(id) \models o \quad d(\Delta, id) = id' \quad \Delta(id') \models i \qquad \Delta' = \mathtt{upd}(id, id', f^d, \Delta)}{\Delta; (id)([f^d]^o_i \mathrm{P} + \mathrm{S}) \to \Delta'; [f^d]^o_i((id')\mathrm{P}) + \mathrm{S}} \ (\textsc{Bin})$$

$$\frac{\Delta(id) \models o \quad \star(\Delta, id) = I' \quad I = \{id' \mid id' \in I' \wedge \Delta(id') \models i\}}{\Delta; (id)([f^\star]^o_i \mathrm{P} + \mathrm{S}) \to \Delta'; [f^\star]^o_i((I)\mathrm{P}) + \mathrm{S}} \ (\textsc{Brd})$$

$$\frac{\Delta(id) \models o \quad \Delta(id) \models i}{\Delta; (id)([f^\bullet]^o_i \mathrm{P} + \mathrm{S}) \to \Delta; [f^\bullet]^o_i((id)\mathrm{P}) + \mathrm{S}} \ (\textsc{Loc})$$

$$\frac{\Delta; \mathrm{P} \to \Delta'; \mathrm{P}'}{\Delta; [f^d]^o_i \mathrm{P} \to \Delta'; [f^d]^o_i \mathrm{P}'} \ (\textsc{Synch}) \qquad \frac{\Delta; \mathrm{P} \to \Delta'; \mathrm{P}'}{\Delta; (id)\mathrm{P} \to \Delta'; (id)\mathrm{P}'} \ (\textsc{Id})$$

$$\frac{\Delta; \mathrm{P}_1 \to \Delta'; \mathrm{P}_1'}{\Delta; \mathrm{P}_1 + \mathrm{P}_2 \to \Delta'; \mathrm{P}_1' + \mathrm{P}_2} \ (\textsc{Sum}) \qquad \frac{\Delta; \mathrm{P}_1 \to \Delta'; \mathrm{P}_1'}{\Delta; \mathrm{P}_1 \mid \mathrm{P}_2 \to \Delta'; \mathrm{P}_1' \mid \mathrm{P}_2} \ (\textsc{Par})$$

$$\frac{\mathrm{P} \equiv \mathrm{P}' \qquad \Delta; \mathrm{P}' \to \Delta'; \mathrm{Q}' \qquad \mathrm{Q}' \equiv \mathrm{Q}}{\Delta; \mathrm{P} \to \Delta'; \mathrm{Q}} \ (\textsc{Struct})$$

o, as specified in premise $\Delta(id) \models o$. Furthermore, the reacting node id', specified in the premise $d(\Delta, id) = id'$, is required to satisfy condition i. As a result of the synchronisation, the configuration evolves to $[f^d]^o_i((id')\mathrm{P}) + \mathrm{S}$, specifying that id' is active on the continuation protocol P. The resulting network state is obtained by considering the side-effects of the synchronisation. Note that the synchronisation action construct is preserved after the respective synchronisation (see Example 3). We omit the rule that captures the case for the singleton summation (likewise for Brd and Loc).

Rule Brd captures broadcast interaction ($\star$) and is similar to rule Bin. Except for the absence of side-effects, the main difference is that now a set of potential reacting nodes is identified (I' denotes a set of node identifiers), out of which all those satisfying condition i are singled out (I). The latter are activated to carry the continuation protocol. We use (I) to abbreviate $(id_1) \dots (id_m)$ considering $I = id_1, \dots, id_m$. We remark that the set of reacting nodes may be empty (e.g., if none of the potential ones satisfies condition i), in which case $(\emptyset)\mathrm{P}$ is defined as P. Note that the reduction step nevertheless takes place, even without reacting nodes, modelling a non-blocking broadcast. This differs from the binary interaction which is blocked until all conditions are met, including the reacting one.

Rule Loc captures local computation steps ($\bullet$). For the sake of uniformity we keep (both) output and input conditions. Note that the node that carries out the f step retains control, i.e., the same id active in f is activated in the continuation P. Like for broadcast, we consider local steps do not involve any state update.

Rules for language closure state that nodes can be active at any stage of the protocol, hence reduction may take place at any level, including after a synchronisation action (rule Synch) and within a summation (rule Sum). By preserving the structure of the protocol, including synchronisation actions that have been carried out, we allow for participants to be active on (exactly) the same stage of the protocol simultaneously and at different moments in time, as the next example shows. Rules Id and Par follow the same principle and finally, rule Struct closes reduction under structural congruence.

Example 3. Assuming that node id_2 satisfies conditions i_2 and o_2 in Δ we may derive, using axiom Loc, rule Sum, rule Struct, and rule Id the reduction:

$$\Delta; (id_1)(id_2)([f_1{}^{\star}]_{i_1}^{o_1}\mathrm{P}_1 + [f_2{}^{\bullet}]_{i_2}^{o_2}\mathrm{P}_2) \to \Delta; (id_1)([f_1{}^{\star}]_{i_1}^{o_1}\mathrm{P}_1 + [f_2{}^{\bullet}]_{i_2}^{o_2}(id_2)\mathrm{P}_2)$$

where node id_2 carries out the f_2 local action. Notice that node id_1 is still active on the summation protocol, and both synchronisations are possible regardless of the summation branch involved in the reduction step involving id_2 shown above.

Since we are interested in developing protocols that may be used in different networks, we will focus on static protocols for the purpose of the development, where static protocols are given by the (id)-free fragment of the language. Then, to represent a concrete operating system, active nodes may be added at "top-level" to the static specification (e.g., $(id)\mathrm{P}_s$ where P_s denotes a static specification, hence does not specify any active nodes), together with a concrete network state (e.g., $\Delta; (id)\mathrm{P}_s$).

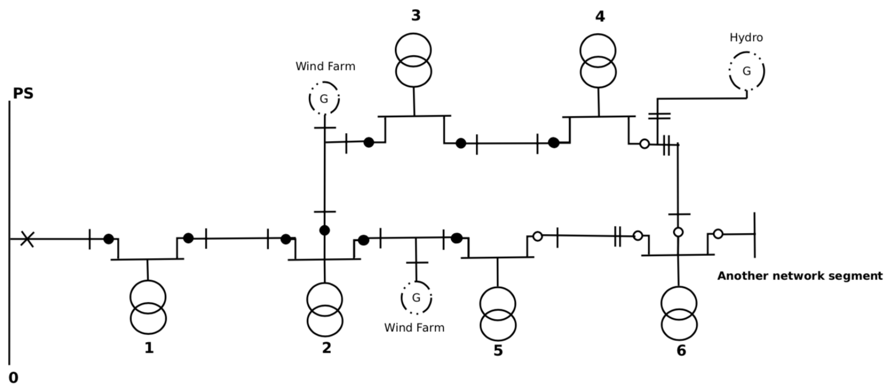
Also, to simplify protocol design, we consider that action labels are unique (up to recursion unfolding) and that, as usual, recursion is guarded by (at least) one synchronisation action (excluding, e.g., **rec** $X.X$). In the remainder, we only consider well-formed protocols that follow the above guidelines, namely: originate from specifications where recursion is guarded, all action labels are distinct, and where the active node construct only appears top-level (e.g., $(id_1)\dots(id_k)\mathrm{P}_s$).

3 Management of Distributed Generation in Power Grids

In this section, we model a protocol for managing distributed generation when major faults in power sources happen. The goal is to find a replacement for failed power sources and transfer the control to it.

We consider a cross section of a radial network of a power grid in Fig. 1. The network consists of a primary power substation **PS**, two generators Hydro and Wind Farms, and six secondary power substations, numbered from **1** to **6**. The type of this network is called radial because every substation has only one incoming power input and possibly multiple power outputs. Each secondary substation has fault indicators (fault $\bullet$ and no fault $\circ$), line switches (closed $|$ and open $\|$), and an embedded controller that implements the substation's behaviour and manages interactions with others. Fig. 1 illustrates a configuration where the secondary substations **1-5** are energised by the primary substation **PS**, while substation **6** is energised by Wind Farms. Secondary substations cannot operate the switches or exchange information without authorisation from the primary substation which supplies the power.

Fig. 1: Power Distribution Grid



Let us consider that the power source of the primary substation **PS** failed which caused a blackout in its domain. The substation **PS** initiates a reconfiguration protocol by synchronising with its directly connected secondary substations and delegates them to locate the substation managing the hydroelectric generator so to transfer the control to it and restore power. To simplify the presentation, our reconfiguration protocol is designed specifically to handle the configuration in Fig. 1. However, it can be easily extended to handle any configuration.

Every substation delegates the substations connected to its output power lines to collaborate to locate the generator. Once the signal is received by the substation managing the generator (in our case, substation **4**), it reconfigures itself, triggers the generator, and the relocation protocol starts. A swap signal is propagated in the direction of **PS** to reconfigure the connections of secondary substations to the direction of the new source. Once this signal is received by **PS**, it sends a permit signal to the substation where the signal came from so that the control is transferred to the new source. At this point the transference protocol starts and a release signal is propagated to the new source. Once received, the new source claims the manager role and retains the control.

We fix the following terminology before we model the protocol in our global language: the state of a source link t can be 0 (to indicate a faulty link, i.e., no power) or 1 otherwise. We use z as the source id when a substation is not connected to a power supply. We also use ∞ in place of the source for all primary stations. Initially, substations, with direct links to backup generators, e.g., substation **4**, record the generators identities in their states regardless of their current sources; otherwise the generation field is reset to $\perp$, e.g., substation **3**. The initial state of each substation follows from Fig. 1. For instance, substations **3**, **4**, and **PS** have the following initial states $\mathbf{3}[2, 0, \{2, 4\}, 1, 1, 0, \perp]$, $\mathbf{4}[3, 0, \{3, 6\}, 1, 0, 0, H]$, and $\mathbf{PS}[\infty, 0, \{1\}, 2, 1, 0, \perp]$, respectively.

The reconfiguration protocol is reported below:

$$\textsc{Reconfiguration} \triangleq [\mathsf{Unlink}^{\star}]^{o_1}_{i_1} \mathbf{rec}\ X.([\mathsf{Locate}^{\star}]^{o_2}_{i_1} X + [\mathsf{Found}^{\blacktriangle}]^{o_3}_{i_1} \textsc{Relocation})$$

The protocol states that Unlink is broadcasted to the children of the enabling substation, after which a recursive protocol is activated on the children. The latter states that either Locate is broadcasted to the children of the enabling substation, after which the recursive protocol starts over, or Found is sent to the parent which consequently carries out the Relocation protocol.

A substation enabled on Reconfiguration enables Unlink only when its source link is faulty and it serves as a primary substation, i.e., $o_1 = (t = 0) \wedge (i = \infty)$. Furthermore, a reactive/receiving substation can always synchronise on Unlink, i.e., $i_1 = \mathsf{true}$.

The children carry out the continuation protocol which is responsible for finding a replacement power source and guaranteeing safe reconfiguration of the network. A child can broadcast Locate only if it cannot serve as a replacement power source, i.e., $o_2 = (g \neq H)$; otherwise, when $o_3 = (g = H)$, Found is sent to the parent. Note that Locate has no side-effects on both sides while Found requires that the enabling station sets its source to ∞ and the receiver sets its generation field (g) to the *id* of the enabling substation. Note that once Found is executed, the substation triggers the hydroelectric generator to supply power and marks itself as a new replacement. The power will be restored once the last substation in the network segment is configured correctly. Furthermore, both actions can always be received. The relocation protocol is reported below:

$$\textsc{Relocation} \triangleq \mathbf{rec}\ Y.([\mathsf{Swap}^{\blacktriangle}]^{o_4}_{i_1} Y + [\mathsf{Permit}^{\blacktriangleright}]^{o_5}_{i_2} \textsc{Transference})$$

Except for the primary substation, any substation enabled on Relocation can send Swap, i.e., $o_4 = (i \neq \infty)$; otherwise, when $o_5 = (i = \infty)$, Permit is sent to its neighbour which would carry out the Transference protocol. The receiver can always synchronise on Swap while for Permit the generation field (g) and the source (i) of the substation should be equal, i.e., $i_2 = (i = g)$. Swap requires that senders sets the value of their source to the one of their generation field. The side-effects of Swap on the receiver are the same as of Found. For Permit, the enabling station disconnects itself and sets its source to z. Furthermore, Permit requires that the receiver resets its generation field. The Transference protocol is reported below:

$$\textsc{Transference} \triangleq \mathbf{rec}\ Z.([\mathsf{Release}^{\blacktriangle}]^{o_6}_{i_1} Z + [\mathsf{Claim}^{\bullet}]^{o_7}_{i_1} \textsc{Reconfiguration})$$

Similarly, the Transference protocol propagates a Release signal to release the manager role of the network segment to the new source. Once the signal is received, the new substation declares the end of the protocol by enabling Claim which is a local signal. This way, the substation retains the control and becomes ready to carry out the whole Reconfiguration protocol. Except for the replacement substation, any substation can enable Release, i.e., $o_6 = (g \neq H)$; otherwise, when $o_7 = (g = H)$, Claim is enabled. Note that Release only requires that the receiver resets its generation field while Claim has not side-effects at all.

Fig. 2: Target Language Syntax

Tests	$T ::=$	$T \mid T$	parallel
	$\mid$	$\mathbf{0}$	termination
	$\mid$	$\mathbf{rec}\ X.T$	recursion
	$\mid$	X	recursion variable
	$\mid$	$\Sigma_{i \in I}\, \alpha_i?.T_i$	input summation
	$\mid$	$\alpha!.T$	output
	$\mid$	$\surd$	success
Action prefix	$\alpha ::=$	$\langle c \rangle f^d$	

The static protocol Reconfiguration abstracts from the concrete network configuration. To represent a concrete network, active substations identifiers must be added at "top-level" together with a network state Δ, i.e., Δ; $(\mathbf{PS})$Reconfiguration. Note that the primary station, **PS**, is initially active because in our scenario it is the only station that can initiate the protocol.

4 Testing for Protocol Fidelity

In this section, we present a technique ensuring that individual node controllers follow a global protocol specification. We start by introducing a testing language that provides a means of interaction with an implementation. Implementations are left abstract as our focus is on the verification technique that only relies that such implementations exhibit determined actions. Then, we show how we can synthesise testers out of a global protocol specification, building on which we introduce a notion of protocol compliance that characterises implementations that pass the synthesised tests. We then present our protocol fidelity result (Theorem 1) that attests implementations compliant with a protocol exhibit the local actions prescribed by the global specification.

The syntax of tests is given in Figure 2. A test can be a parallel composition $T \mid T$, the terminated process $\mathbf{0}$, the recursive definition $\mathbf{rec}\ X.T$, a recursion variable X, an input summation $\Sigma_{i \in I}\, \alpha_i?.T_i$, an output $\alpha!.T$ and the success $\surd$. We remark that since we are interested in interacting with an implementation, we do not expect interaction among different tests. Hence, a parallel composition of tests captures two simultaneously active tests, but where no interaction can occur, as will be made clear by the semantics of the language. Furthermore, we remark on the $\surd$ introduced for the sake of signalling the success of a test. Finally, notice that action prefixes ($\langle c \rangle f^d$) are defined so as to match the observables expected in the global interaction, identifying the synchronisation label f, the direction d and the condition c that either refers to input or output conditions.

The semantics of testers is defined in terms of the following observable actions: $\lambda ::= \alpha! \mid \alpha? \mid \surd$. An observation λ can then be an input or an output or a success label. We then define the semantics of tests by the rules given in Fig. 3, which we now briefly discuss. Rule ParL allows the left part of a parallel composition

Fig. 3: Tester Language LTS

$$\frac{T_1 \xrightarrow{\lambda} T_1' \quad \lambda \neq \surd}{T_1 \,|\, T_2 \xrightarrow{\lambda} T_1' \,|\, T_2}\ \textsc{ParL} \qquad \frac{T_1 \xrightarrow{\surd} T_1' \quad T_2 \xrightarrow{\surd} T_2'}{T_1 \,|\, T_2 \xrightarrow{\surd} T_1' \,|\, T_2'}\ \textsc{ParS}$$

$$\frac{T[\mathbf{rec}\ X.T/X] \xrightarrow{\lambda} T'}{\mathbf{rec}\ X.T \xrightarrow{\lambda} T'}\ \textsc{Rec} \qquad \frac{}{\surd \xrightarrow{\surd} \mathbf{0}}\ \textsc{Success}$$

$$\frac{j \in I}{\Sigma_{i\in I}\, \alpha_i?.T_i \xrightarrow{\alpha_j?} T_j}\ \textsc{Input} \qquad \frac{}{\alpha!.T \xrightarrow{\alpha!} T}\ \textsc{Output}$$

Fig. 4: Testing Semantics

$$\frac{I \xrightarrow{\alpha!} I' \quad T \xrightarrow{\alpha?} T'}{I \parallel T \xrightarrow{\tau} I' \parallel T'}\ \textsc{CommL} \qquad \frac{I \xrightarrow{\tau} I'}{I \parallel T \xrightarrow{\tau} I' \parallel T}\ \textsc{Internal}$$

to evolve by itself by showing a label different from the success ($\surd$) one. The symmetric rule for the right part of the parallel composition is omitted. The only way a success label can be propagated through a parallel composition is when both parts are able to produce such label as reported in ParS. Rule Rec deals with recursive processes in a standard way. Rule Success allows a success prefix to reduce to the idle process, while rules Input and Output show how the prefixes exhibit the corresponding labels and activate the continuation. We say test T succeeds, written $T \downarrow_{\surd}$, if $T \xrightarrow{\surd} T'$ for some T'.

As one can notice, in the given LTS there is no rule for synchronisation. As previously announced, this is due to our goal of testing implementations, so the goal is to allow testers to interact with an implementation and not among themselves. We abstract away from how implementations are defined, and consider implementations, ranged over by I, as *black-boxes* that exhibit labels of the form $\alpha!$, $\alpha?$ and τ. For the purpose of testing an implementation, we define a new level of semantics given by the rules in Figure 4, describing the interactions between a test T and an implementation I, where we use $\parallel$ to specify the parallel composition operator for the testing level. We consider that the actions of implementations and tester are identical (up to the represented duality), which in particular means that the conditions are exactly the same. Considering logical equivalence instead would be more appropriate to support more flexibility, but for the sake of simplifying the presentation we adopt here syntactic equality. We leave for future work the refinement of this notion, together with a more in depth exploration of the possible logical support for the correspondence, and consider here that implementations refer to conditions as specified in the protocol.

The rules of Figure 4 on the one hand capture the interaction between test and implementation (rule CommL and the omitted symmetric version), and on

Table 4: Tester Synthesis

$[\![[f^d]^o_i P]\!]^?_\sigma$	$\triangleq \langle i \rangle f^d!.[\![P]\!]^!_\sigma \mid [\![P]\!]^?_\sigma$		iSynch
$[\![[f^d]^o_i P]\!]^!_\sigma$	$\triangleq \langle o \rangle f^d?.\surd$		oSynch
$[\![\mathbf{0}]\!]^{\mathtt{r}}_\sigma$	$\triangleq \mathbf{0}$		pNil
$[\![X]\!]^?_\sigma$	$\triangleq \mathbf{0}$		pVar
$[\![X]\!]^!_\sigma$	$\triangleq [\![P]\!]^!_\sigma$	$(\sigma(X) = P)$	oVar
$[\![\mathbf{rec}\ X.P]\!]^{\mathtt{r}}_\sigma$	$\triangleq [\![P]\!]^{\mathtt{r}}_{\sigma[X \mapsto P]}$		pRec
$[\![P \mid Q]\!]^{\mathtt{r}}_\sigma$	$\triangleq [\![P]\!]^{\mathtt{r}}_\sigma \mid [\![Q]\!]^{\mathtt{r}}_\sigma$		pPar
$[\![S_1 + S_2]\!]^!_\sigma$	$\triangleq [\![S_1]\!]^!_\sigma + [\![S_2]\!]^!_\sigma$		oSum
$[\![S_1 + S_2]\!]^?_\sigma$	$\triangleq [\![S_1]\!]^?_\sigma \mid [\![S_2]\!]^?_\sigma$		pSum

the other hand abstract away from the implementation internal behaviours (rule Internal). The latter, conceivably, can be further generalised by disregarding actions that are not relevant for the particular tester considered, e.g., by identifying the set of labels of relevant actions and, like for τ, allowing evolutions of the implementation that carry a non-relevant action label to be interleaved.

For the purpose of defining protocol compliance, we rely on the traces observed for the composition of an implementation and a test. In order to abstract from the internal moves of implementations, we rely on the weak variant of the transitions, defined next following standard lines. As usual, we add τ to the set of relevant observations (because protocols also involve local steps) and use λ^τ to represent either a λ or τ. We then denote by $I \parallel T \overset{\lambda^\tau}{\Longrightarrow} I' \parallel T'$ the evolution from $I \parallel T$ to $I' \parallel T'$ comprising a (possibly empty) sequence of τ steps and a λ^τ, hence $I \parallel T \xrightarrow{\tau} \cdots \xrightarrow{\tau}\xrightarrow{\lambda^\tau}\xrightarrow{\tau} \cdots \xrightarrow{\tau} I' \parallel T'$. Also, we denote by $I \parallel T \overset{\tilde{\lambda}^\tau}{\Longrightarrow} I_n \parallel T_n$ the sequence $I \parallel T \overset{\lambda^\tau_1}{\Longrightarrow} I_1 \parallel T_1 \overset{\lambda^\tau_2}{\Longrightarrow} \cdots \overset{\lambda^\tau_n}{\Longrightarrow} I_n \parallel T_n$ when $\tilde{\lambda}^\tau = \lambda^\tau_1, \cdots, \lambda^\tau_n$.

We now show how we can automatically generate testers out of a protocol specification. Tests T are synthesised directly from a protocol P through the function $[\![P]\!]^{\mathtt{r}}_\sigma$ defined in Table 4, where σ is a mapping from recursion variables to protocols and $\mathtt{r}$ identifies the type of projection. When $\mathtt{r}$ is ? the result of the projection tests if the implementation has an (expected) input. On the other hand when $\mathtt{r}$ is ! the result of the projection tests if the implementation has an (expected) output. The result of the (combination of these two types of) projection allows one to verify *static* protocols (cf. Section 2).

We briefly discuss the definition of projection. In case iSynch, the projection yields the output ($\langle i \rangle f^d!.[\![P]\!]^!_\sigma$) that is intended to interact with the expected corresponding input. For this purpose, notice that the condition specified in the tester output is precisely the one expected for the input i. Also, the continuation of the tester output ($[\![P]\!]^!_\sigma$) checks if the implementation can afterwards (i.e., after the input) exhibit the active behaviour of the continuation. Therefore the type of the projection for the continuation is !, and hence will test the implementation exhibits the expected outputs. Finally, the remainder of the protocol is (inductively) projected for generating testers for other inputs ($[\![P]\!]^?_\sigma$), which are collected in parallel so inputs are tested without a causality relation, while the causality is present between the input and the output reactions.

In case oSynch the yielded tester input specifies the condition of the output expected from the implementation. If the implementation matches the expectancy then the synchronisation may occur, in which case (this part of) the test succeeds and hence the continuation of the tester input is $\surd$. The remaining cases show how the two types of projections inductively proceed in the structure of the protocol so as to generate the tester inputs and outputs for the whole of the protocol. We remark that σ is used to generate the ! projection when a recursion variable occurs in the continuation of a synchronisation action (cf., pRec and oVar).

We may now define the notion of protocol compliance in a way similar to the notion of passing a test [10]. Protocol compliance relies on the ?-projection to check if all expected inputs may be exhibited by the implementation, while ?-projection relies on !-projection to check for the expected outputs.

Definition 1 (Protocol Compliance). *We say implementation I is compliant with protocol P, written $(I \parallel P) \Downarrow_{\surd}$, if*

$$I \parallel [\![P]\!]^{?}_{\emptyset} \overset{\tau}{\Rightarrow} I' \parallel T' \qquad \textit{and} \qquad T' \downarrow_{\surd}$$

The key idea of protocol compliance is to rely on an extensional observational characterisation which allows to abstract away from implementation details.

The compositionality principle stated next is of particular use in our setting, considering different protocols are developed using different action label alphabets (cf. well-formed protocols). We remark that the projection is conservative in this respect, i.e., protocols with disjoint action label alphabets yield testers that also have disjoint action label alphabets. We then say that two tests T_1 and T_2 are non-interfering if the sets of their action prefixes are disjoint, denoted by $T_1 \# T_2$.

Proposition 1 (Compositionality). *If I passes tests T_1 and T_2 with $T_1 \# T_2$, then I passes test $T_1 \,|\, T_2$.*

Proposition 1 thus allows to focus the verification of implementations considering each protocol separately, given an implementation that is compliant with two protocols individually will also be compliant with the (parallel) combination of the protocols.

As mentioned previously, protocol compliance addresses static protocols, hence ensures that implementations exhibit all expected reactions. For the purpose of our protocol fidelity result, we need to have a means of specifying the initial enabling behaviour that is introduced by adding to a static protocol active nodes at top-level. To this end, we define a third kind of projection, shown in Table 5, that yields the outputs corresponding to the top-level actions of a protocol.

Example 4. To illustrate the synthesis of a test from a protocol specification, we consider a simplified version of the Reconfiguration protocol in Section 3 as follows:

$$\text{Reconfiguration} \triangleq [\mathsf{Unlink}^{\star}]^{o_1}_{i_1} \mathbf{rec}\ X.([\mathsf{Locate}^{\star}]^{o_2}_{i_1} X + [\mathsf{Found}^{\blacktriangle}]^{o_3}_{i_1} \mathbf{0})$$

Table 5: Enabler Synthesis

$[\![[f^d]_i^o P]\!]_\sigma^e$	$\triangleq \langle o\rangle f^d!.\mathbf{0}$		iSynch
$[\![\mathbf{0}]\!]_\sigma^e$	$\triangleq \mathbf{0}$		iNil
$[\![X]\!]_\sigma^e$	$\triangleq [\![P]\!]_\sigma^e$	$(\sigma(X) = P)$	iVar
$[\![\mathbf{rec}\ X.P]\!]_\sigma^e$	$\triangleq [\![P]\!]_{\sigma[X\mapsto P]}^e$		iRec
$[\![P \mid Q]\!]_\sigma^e$	$\triangleq [\![P]\!]_\sigma^e \mid [\![Q]\!]_\sigma^e$		pPar
$[\![S_1 + S_2]\!]_\sigma^e$	$\triangleq [\![S_1]\!]_\sigma^e + [\![S_2]\!]_\sigma^e$		iSum

The e-synthesized test according to Table 5 is:

$$\langle o_1\rangle \mathsf{Unlink}^\star!.\mathbf{0}$$

The ?-synthesized test according to Table 4 is:

$$\begin{array}{l}\langle i_1\rangle \mathsf{Unlink}^\star!.(\langle o_2\rangle \mathsf{Locate}^\star?.\surd\ +\ \langle o_3\rangle \mathsf{Found}^\blacktriangle?.\surd)\ \mid \\ \langle i_1\rangle \mathsf{Locate}^\star!.(\langle o_2\rangle \mathsf{Locate}^\star?.\surd\ +\ \langle o_3\rangle \mathsf{Found}^\blacktriangle?.\surd)\ \mid\ \langle i_1\rangle \mathsf{Found}^\blacktriangle?.\mathbf{0}\end{array}$$

At this point we have all the technical ingredients on the implementation side that allow to characterise protocol fidelity. However, we need to revisit the semantics of the global language, instrumenting it in a way so that evolutions (reductions) carry the respective information (in labels). Namely, we introduce labels that reveal the interacting parties and the synchronisation action that triggered the reduction step. Such labels thus refer to both parties involved in an interaction, while our purpose is to ensure that the implementation of each one of such parties exhibits the prescribed behaviours. So, we need a means to focus a global label on an individual party. Furthermore, we introduce traces of global (labeled) reductions and define a way to trim such traces so as to focus on the contributions of a specific party. All of the above are defined next.

Definition 2 (Labeled Reduction). *Given a reduction $\Delta; P \rightarrow \Delta'; P'$ derived using the rules of Table 3 we write*

1. *$\Delta; P \xrightarrow{(id)[f^d]_i^o(id')} \Delta'; P'$ if the derivation has axiom* Bin*;*
2. *$\Delta; P \xrightarrow{(id)[f^d]_i^o(\tilde{I})} \Delta'; P'$ if the derivation has axiom* Brd*;*
3. *$\Delta; P \xrightarrow{\tau} \Delta'; P'$ if the derivation has axiom* Loc*.*

We use ξ to range over such labels.

We now define the operation that allow us to focus global protocol labels on an individual participant. Given a protocol label ξ we define $(\xi)\lightning_{id}$ as follows:

$$\begin{array}{llll}((id)[f^d]_i^o(id'))\lightning_{id} & = !\langle o\rangle f^d & ((id')[f^d]_i^o(id))\lightning_{id} & = ?\langle i\rangle f^d \\ ((id)[f^d]_i^o(\tilde{I}))\lightning_{id} & = !\langle o\rangle f^d & ((id')[f^d]_i^o(\tilde{I}))\lightning_{id} & = ?\langle i\rangle f^d \quad \text{if} \quad id \in \tilde{I} \\ (\tau)\lightning_{id} & = \tau & (\xi)\lightning_{id} & = \epsilon \qquad \text{otherwise}\end{array}$$

We also extend the definition of $(\cdot)\lightning_{id}$ for a trace $\tilde{\xi}$ as $(\tilde{\xi})\lightning_{id} = (\xi)\lightning_{id} \cdot (\tilde{\xi}')\lightning_{id}$ when $\tilde{\xi}$ is $\xi, \tilde{\xi}'$, using '$\cdot$' to represent trace concatenation and taking ϵ as the idempotent element (empty trace). We may now state our main result.

Theorem 1 (Protocol Fidelity). *Let Δ be a network state, id a node identifier, P a protocol and I an implementation such that $(I \parallel P) \Downarrow_{\surd}$. We have that:*

$$\begin{array}{ll} \textit{if} & \Delta; (id)P \xrightarrow{\xi_1} \Delta_1; P_1 \xrightarrow{\xi_2} \cdots \xrightarrow{\xi_n} \Delta_n; P_n \\ \textit{then} & I \mid [\![P]\!]^e_{\emptyset} \overset{\tilde{\lambda}}{\Rightarrow} I' \mid T' \quad \textit{with} \quad (\tilde{\xi})\lightning_{id} = \tilde{\lambda} \end{array}$$

Notice that in Theorem 1 we use the e type of projection to inject the initial behaviours correspondent to the ones obtained by the top-level active node. This is because the implementation I is ensured to exhibit the actions of the static part of the protocol P but not the enabling actions corresponding to $(id)P$. So we consider the implementation is composed (in parallel) with the implementation, where synchronisation between them is not supported (hence, parallel supports interleaving here). The result ensures that any sequence of actions prescribed for any node at the level of the global trace is matched by the corresponding actions of the implementation composed with the initial enabling behaviour.

Example 5. We return to the protocol shown in Example 4, namely:

$$\textsc{Reconfiguration} \triangleq [\mathsf{Unlink}^{\star}]^{o_1}_{i_1} \mathbf{rec}\ X.([\mathsf{Locate}^{\star}]^{o_2}_{i_1} X + [\mathsf{Found}^{\blacktriangle}]^{o_3}_{i_1} \mathbf{0})$$

Let I be an implementation such that $(I \parallel \textsc{Reconfiguration}) \Downarrow_{\surd}$. Considerin the e-synthesized test shown in Example 4 we have that Theorem 1 ensures that $I \mid \langle o_1 \rangle \mathsf{Unlink}^{\star}!.\mathbf{0}$ can exhibit the actions corresponding to the reductions of $\Delta; (id)\textsc{Reconfiguration}$. Notice that the initial action of id is given by the e-projection, while remaining actions will be exhibited by the implementation since it complies to the test given by the ?-projection (see Example 4).

We remark that the inverse direction of the implication stated in Theorem 1 does not hold in general considering the protocol compliance given in Definition 1. Intuitively, consider that implementations can exhibit more actions than the ones prescribed by the tests, e.g., an implementation can exhibit simultaneously (in parallel) two actions while the corresponding test prescribes that they must happen in sequence. The strict correspondence is naturally a desirable property that we leave for future work. We also remark that we focus here on observable actions and do not introduce the state information explicitly (which may be separated from the operational implementation). Refining the statement so as to consider explicitly the state information may allow us to abstract away from the logical conditions currently under consideration in the testing that supports the protocol compliance, and explore different notions of logical support for assessing when implementations meet the specifications.

5 Concluding remarks

Ensuring that implementations meet specifications is of crucial importance in software development. Techniques used to guarantee such correspondence should be flexible enough to allow for a great number of implementations to (safely) match a specification, so as to promote their usability. For the purpose of analysing interacting systems, reasoning in terms of observational equivalences has been used ever since the seminal work of Milner (cf. [17]). The key idea is that systems are deemed equivalent if an external observer cannot distinguish between them. Testing [10] embeds this principle and seems particularly fit for the purpose of ensuring that implementations meet specifications, given that the testers may be crafted so as to faithfully represent the specifications. The idea is that two implementations are equivalent if no specification can distinguish between them.

The above principle is at the basis of the development presented in this paper. The goal is to allow for several implementations to conform to a protocol specification, abstracting away from details that do not compromise the safe operation of the system. We have merely scratched the surface of the advantages of using testing in this setting, in particular when taking into account the broadness of the related literature (e.g., [3–5,8,11–14,16,18,19]). However, we can already state a protocol fidelity result that ensures compliant implementations exhibit the actions prescribed by the protocol specifications. Introducing a notion of testing preorder, relating implementations that pass all the tests of other implementations, we may also characterise a substitution principle for the safe replacement of controller implementations.

Our global protocol language can be anchored to the proposal of choreographic programming [6], in the sense that programming is carried out directly at the protocol language level, and operationally correspondent distributed implementations can be automatically generated from the global specification (cf. [2]). We take a different perspective here, admitting that node controllers are developed as usual in a separate way with respect to the specification. In fact, we view implementations in an opaque way so as to allow for greater generality, e.g., allowing for implementations that interleave their participation in different protocols.

Nevertheless, following lines similar to previous work [2], given a protocol specification we may consider a distributed network where each node is equipped with a (compliant) controller implementation. Then, we may also show that the yielded distributed model operationally corresponds to the global (centralised) model of Section 2.

A tribute to Rocco De Nicola. This paper is a contribution to the Festschrift that celebrates Rocco De Nicola's 65th birthday. We tried to gather topics in which Rocco has been a pioneer and a prolific author: *coordination models* and *testing equivalences*. Coordination is the goal of the model presented in Section 2 while Section 4 is undoubtedly inspired by Rocco's seminal work on testing pre-orders [7, 10]. Besides being a very prolific and influential scientist, Rocco has also been a mentor and a source of inspiration for many researchers. His dedication to research and his quest for scientific rigour will inspire generations.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE'16. pp. 1–18 (2016)
2. Alrahman, Y.A., Vieira, H.T.: Operation control protocols in power distribution grids. CoRR abs/1811.01942 (2018)
3. Bernardi, G., Hennessy, M.: Mutually testing processes. Logical Methods in Computer Science 11(2) (2015)
4. Boreale, M., De Nicola, R.: Testing equivalence for mobile processes. Inf. Comput. 120(2), 279–303 (1995)
5. Boreale, M., De Nicola, R., Pugliese, R.: Trace and testing equivalence on asynchronous processes. Inf. Comput. 172(2), 139–164 (2002)
6. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13. pp. 263–274. ACM (2013)
7. De Nicola, R.: Testing equivalences and fully abstract models for communicating systems. Ph.D. thesis, University of Edinburgh, UK (1986), `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.649251`
8. De Nicola, R.: Extensional equivalences for transition systems. Acta Inf. 24(2), 211–237 (1987)
9. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
10. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
11. De Nicola, R., Hennessy, M.: CCS without tau's. In: TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, 1987. LNCS, vol. 249, pp. 138–152. Springer (1987)
12. De Nicola, R., Melgratti, H.C.: Multiparty testing preorders. In: Trustworthy Global Computing - 10th International Symposium, TGC 2015, Revised Selected Papers. LNCS, vol. 9533, pp. 16–31. Springer (2015)
13. De Nicola, R., Segala, R.: A process algebraic view of input/output automata. Theor. Comput. Sci. 138(2), 391–423 (1995)
14. Hennessy, M.: Algebraic theory of processes. MIT Press series in the foundations of computing, MIT Press (1988)
15. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. 49(1), 3:1–3:36 (2016)
16. Laneve, C., Padovani, L.: The *Must* preorder revisited. In: Concurrency Theory, 18th International Conference, CONCUR 2007, Proceedings. LNCS, vol. 4703, pp. 212–225. Springer (2007)
17. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
18. Natarajan, V., Cleaveland, R.: Divergence and fair testing. In: Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Proceedings. LNCS, vol. 944, pp. 648–659. Springer (1995)
19. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. 205(2), 125–198 (2007)