A Predicate Transformer for Choreographies (Full Version)

A Predicate Transformer for Choreographies (Full Version)

Sung-Shik Jongmans[1,2](✉) and Petra van den Bos[3]

[1] Department of Computer Science, Open University, Heerlen, the Netherlands
[2] CWI, Amsterdam, the Netherlands
[3] Formal Methods and Tools Group, University of Twente, Enschede, the Netherlands

Abstract. Construction and analysis of distributed systems is difficult; choreographic programming is a deadlock-freedom-by-construction approach to simplify it. In this paper, we present a new theory of choreographic programming. It supports for the first time: construction of distributed systems that require decentralised decision making (i.e., if/while-statements with multiparty conditions); analysis of distributed systems to provide not only deadlock freedom but also functional correctness (i.e., pre/postcondition reasoning). Both contributions are enabled by a single new technique, namely a predicate transformer for choreographies.

1 Introduction

Construction and analysis of distributed systems that consist of message passing processes is hard. Typical challenges include providing *deadlock freedom* (i.e., the processes never get stuck) and *functional correctness* (i.e., the processes compute the intended outcome). *Choreographic programming* [8,9,10] is a deadlock-freedom-by-construction approach to make implementation and verification of distributed systems easier. In this paper, to address two limitations of existing theories, we present a new theory of choreographic programming. It supports for the first time: construction of distributed systems that require **decentralised decision making**; analysis of distributed systems to provide not only deadlock freedom but also **functional correctness**.

1.1 Background: Choreographic Programming by Example

To explain choreographic programming, consider a distributed system in which two processes enact *roles* Client and Server. First, a username and password are communicated from Client to Server. Next, Server checks Client's credentials and informs Client about the outcome: if authentication succeeded, the execution continues; if it failed, it ends. We construct and analyse this system as follows:

1. Initially, we write a **global program** G ("the choreography"); it prescribes the behaviour of all roles, collectively, from their shared perspective.

$$\texttt{C."foo"} \twoheadrightarrow \texttt{S.x} \,;\, \texttt{C.123} \twoheadrightarrow \texttt{S.y} \,;\, \textbf{if}\ \texttt{S.auth(x,y)}\ (\texttt{S.SUCC} \twoheadrightarrow \texttt{C} \,;\, G')\ (\texttt{S.FAIL} \twoheadrightarrow \texttt{C})$$

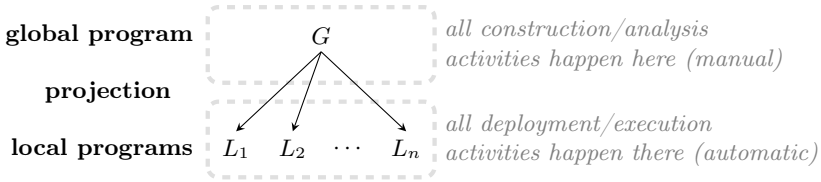

Fig. 1: Workflow of choreographic programming

In this notation, $p.e \twoheadrightarrow q.y$ prescribes a value communication to share data from role p to role q: expression e is evaluated at p, sent at p, received at q, and stored in variable y at q. Similarly, $p.\ell \twoheadrightarrow q$ prescribes a label communication to share decisions: label ℓ is actively selected at p ("internal choice"), sent at p, received at q, and passively branched on at q ("external choice"). Furthermore, $G_1 \,;\, G_2$ and **if** $r.e\; G_1\; G_2$ prescribe a sequence and a conditional choice (i.e., if e is evaluated to `true` at r, then G_1 is executed, or else G_2). Now, informally, the first theorem of choreographic programming is this:

Theorem 1 (Deadlock Freedom). *Every global program is deadlock-free.*

2. Subsequently, we decompose global program G into **local programs** $L_{\mathtt{C}}$ and $L_{\mathtt{S}}$ ("the processes"), using a **projection function**; every local program prescribes the behaviour of one role, individually, from its own perspective.

   Client: `CS!"foo"` ; `CS!123` ; `SC?{SUCC` : $L'_{\mathtt{C}}$, `FAIL` : **skip**`}`
   Server: `CS?x` ; `CS?y` ; **if** `S.auth(x,y)` (`SC!SUCC` ; $L'_{\mathtt{S}}$) (`SC!FAIL`)

   In this notation, $pq!e$ and $pq?y$ prescribe a send and a receive of a value from p to q. Similarly, $pq!\ell$ and $pq?\{\ell_i : L_i\}_{i \in I}$ prescribe a send and a receive of a label (i.e., if ℓ_j is received for some $j \in I$, then L_j is executed).
   Now, informally, the second theorem of choreographic programming is this:

   Theorem 2 (Operational Equivalence). *Every well-formed global program is operationally equivalent to the parallel composition of its projections.*

   "Well-formedness" is a syntactic condition on global programs; we discuss it in more detail later. Here, we just claim that G above is indeed well-formed.

3. Finally, we compose local programs $L_{\mathtt{C}}$ and $L_{\mathtt{S}}$ in parallel ("the distributed system"), by deploying them concurrently, and by executing them at their own pace; as they run, $L_{\mathtt{C}}$ and $L_{\mathtt{S}}$ send and receive messages as prescribed. Now, Thm. 1 and Thm. 2 together entail that $L_{\mathtt{C}}$ and $L_{\mathtt{S}}$ are deadlock-free, by construction, without extra analysis. Figure 1 summarises the workflow.

1.2 Related Work: State of the Art & Open Problems

Early work on choreographic programming was presented by Carbone et al. [8,9] (using binary session types [34]) and by Carbone and Montesi [10] (using

multiparty session types [35]); substantial progress has been made since. For instance, Montesi and Yoshida developed a theory of compositional choreographic programming that supports open distributed systems [41]; Carbone et al. studied connections between choreographic programming and linear logic [11,12,7]; Dalla Preda et al. combined choreographic programming with dynamic adaptation [47,45,46]; Cruz-Filipe and Montesi developed a minimal Turing-complete language of global programs [16,19]; Cruz-Filipe et al. presented a technique to extract global programs from families of local programs ("choreography extraction") [14]; and recently, Giallorenzo et al. studied a correspondence between choreographic programming and multitier languages [29]. Other work on choreographic programming includes results on case studies [15], procedural abstractions [18], asynchronous communication [17], polyadic communication [20,31], implementability [28], and formalisation/mechanisation in Coq [21,22]. Furthermore, theoretical developments are supported in practice by several tools, including Chor [10], AIOCJ [47,46], and Choral [29].

However, all publications cited above have two limitations:

1. Regarding the **construction** of distributed systems, existing work on choreographic programming supports only *centralised* decision making: every if/while-statement in a global program has a *one-party* condition, evaluated at a single role. For instance, in the example above, the decision to continue or end the execution is made by Server alone; Client is duly informed afterwards—with a label communication—as it needs to know how to proceed, but the decision is really Server's.
   However, in many distributed systems, it is *impractical* (i.e., unnecessary or unnatural), or even *impossible*, for a single role to make decisions.
   For instance, consider a distributed system in which two processes enact roles Player1 and Player2 to simulate a game of chess. The idea is that, at the end of every turn, a move is communicated from "active" Player$i$ to "passive" Player$j$, after which a decision must be made: should Player$j$ take a next turn, or is the game over? The key point here is that every role has enough knowledge to check if the latest move is, in fact, the final one. So after every turn, every role can privately—without a label communication—decide to continue or end the execution; moreover, unanimity is guaranteed. It is, thus, unnecessary to *additionally* use a label communication to have one role explicitly inform the other one about how to proceed. Yet, all publications cited above force the usage of a label communication in this situation anyway.
2. Regarding the **analysis** of distributed systems, existing work on choreographic programming focusses on providing deadlock freedom. In contrast, providing functional correctness has not received due attention. This is surprising: given the sequential programming style in which global programs are expressed, it seems worthwhile to study how classical verification techniques for sequential code can be adapted to choreographic programming.

Beyond choreographic programming, all other choreography-based approaches that we know of are limited to centralised decision making, including conversation protocols (e.g., [3,27]), multiparty session types (MPST) (e.g., [35,13,23,24]),

Table 1: State of the art (e.g., [9,10,12,19,29,41,46]) vs. this paper

	state of the art	**this paper**
construction		
decisions	centralised	decentralised
conditions	one-party	multiparty
syntax	**if** $r.e$ G_{then} G_{else}	**if** $\bigwedge\{r.e_r\}_{r \in R}$ G_{then} G_{else}
example (global programs)	1. A.x2 → A.y1 ; 2. **if** A.x1==y1 3. A.SUCC → B ; G_{then} 4. A.FAIL → B ; G_{else}	1. A.x2 → A.y1 ; A.x1 → B.y2 ; 2. **if** A.x1==y1 ∧ B.x2==y2 3. G_{then} 4. G_{else}
analysis	deadlock freedom	deadlock freedom & functional correctness

and MPST extensions to support value-based reasoning using assertions [5], dependent types [51,25], and refinement types [52]. Furthermore, we note that (elements of) deductive verification and session types were combined in Actris [32] and ParTypes [40]. Actris supports reasoning about functional correctness (using separation logic [43,36]), but only for *binary* sessions. In contrast, ParTypes supports multiparty sessions, but it does not consider functional correctness.

1.3 Contributions of This Paper

In this paper, we address the two limitations described in Sect. 1.2.

1. **Construction:** We present a new theory of choreographic programming that supports *decentralised* decision making: every if/while-statement has a *multiparty* condition, evaluated at multiple roles.
2. **Analysis:** The new theory ensures that if the *precondition* is true in the *initial state* of a global program, then after executing the global program, the *postcondition* is true in the *final state*. Similar to deadlock freedom, this form of functional correctness is conferred from the global program to the parallel composition of its projections, by operational equivalence.

Table 1 summarises our contributions relative to the state of the art; it also shows a minimal example to illustrate the essential difference between centralised decision making and decentralised. With centralised decision making (left global program), first, *only Bob* shares `x2` with Alice; next, *only Alice* compares it with `x1` and shares the outcome with Bob. In contrast, with decentralised decision making (right global program), first, *both Alice and Bob* share their values; next, *both Alice and Bob* compare them, but they do not need to share the outcomes, as their unanimity is guaranteed.

1.4 Key Challenge: How to Check If Unanimity Is Guaranteed?

So far, we have seen two examples of decentralised decision making (i.e., Player1 and Player2 in Sect. 1.2; Alice and Bob in Sect. 1.3). In both examples, we noted

that "unanimity is guaranteed"; this is crucially important to provide deadlock freedom. As a counterexample of what can go wrong in the absence of unanimity, suppose that Bob's condition in Tab. 1 were `x2==true` (i.e., he ignores Alice's value). In that case, unanimity is not guaranteed, so Alice and Bob can *diverge*: Alice privately decides to enter one branch, while Bob privately decides to enter the other branch. A deadlock subsequently ensues if, for instance, Alice needs to await a message from Bob in her branch, while Bob needs to await a message from Alice in his branch.

Thus, the key challenge to support decentralised decision making in choreographic programming is this: "How to check if unanimity is guaranteed?" The pivotal insight is that this question can be reduced to a seemingly unrelated one: "Given a global program and a postcondition, how to compute a precondition?" It was first answered for sequential code by Dijkstra in the 1970s [26], in terms of a *predicate transformer* to compute *weakest preconditions*. A crucial technical contribution of this paper is a non-trivial adaptation of Dijkstra's seminal work, tailored for choreographic programming, to provide not only functional correctness (i.e., ensure the truth of the postcondition) but also deadlock freedom in the presence of decentralised decision making (i.e., ensure unanimity).

1.5 Organisation of This Paper

In Sect. 2, to further motivate this paper's new theory, we present more examples of real(istic) distributed systems that require decentralised decision making.

The new theory is presented in Sects. 3–7: in Sect. 3, we present some preliminaries; in Sect. 4, we present a base calculus of global programs, without if/while-statements, but with a main theorem that covers both deadlock freedom and functional correctness; in Sect. 5 and Sect. 6, to support decentralised decision making, we extend the base calculus with if/while-statements; in Sect. 7, we present a calculus of local programs and projection. Thus, Sect. 4–6 cover the upper half of Fig. 1, while only Sect. 7 covers the bottom half.

Detailed definitions, auxiliary lemmas, main theorems, and proofs appear in a technical report [39].

2 Motivating Examples

To further motivate the *usefulness* and *necessity* of this paper's new theory, in this section, we present examples of real(istic) distributed systems that require decentralised decision making; see Appx. A for additional examples. Throughout the section, we adopt a programmer's perspective and present only global programs (i.e., all construction and analysis activities that a programmer carries out manually in the workflow, happen in the upper half of Fig. 1).

Regarding the usefulness of the new theory, the following example shows that centralised decision making can be *impractical* (i.e., unnatural or unnecessary).

Example 1 (Chess simulation). From Sect. 1.2, recall the distributed system in which two processes enact roles Player1 and Player2 to simulate a game of chess.

```
1. P1.b:=board() ; P2.b:=board() ;
2. while P1.!done(b)
3.   (P1.CONTINUE ↠ P2 ; G12 ;
4.    if P2.!done(b)
5.      (P2.CONTINUE ↠ P1 ; G21)
6.      (P2.END ↠ P1 ; skip)) ;
7. P1.END ↠ P2
```

(a) Centralised

```
1. P1.b:=board() ; P2.b:=board() ;
2. while P1.!done(b) ∧ P2.!done(b)
3.   (G12 ;
4.    if P1.!done(b) ∧ P2.!done(b)
5.      G21
6.      skip)
```

(b) Decentralised

Fig. 2: Global programs for chess simulation (Exmp. 1)

Figure 2 shows two global programs: one that uses centralised decision making (at Player1 and Player2, in alternating order), and one that uses the new theory's decentralised decision making; both have auxiliary global programs G_{12} (Player1 is active, Player2 is passive; details omitted) and G_{21} (vice versa).

In Sect. 1.2, we argued for the usefulness of decentralised decision making in this example: the label communications in Fig. 2a are actually unnecessary. □

Regarding the necessity of the new theory, the following example shows that centralised decision making can be *impossible*. In the example, notation $G_1 \parallel G_2$ prescribes an interleaving; it is used to express that the order in which G_1 and G_2 are executed does not matter (i.e., it is not intended to be multi-threading; there is no interaction between G_1 and G_2). By convention, sequencing binds stronger than interleaving. For instance, $G_1 \,;\, G_2 \parallel G_3$ should be read as $(G_1 \,;\, G_2) \parallel G_3$.

Example 2 (Probabilistic leader election in anonymous clique networks). Consider a distributed system in which k anonymous processes (i.e., they have no predefined identifiers) need to elect a leader among them. For clique networks (i.e., each process has a channel to each other process), a probabilistic version of Peleg's algorithm [44] can be used in the style of Itai and Rodeh [37,38]. The algorithm proceeds in rounds. In every round, every process picks a random identifier and sends it to every other process. If there is a unique maximal identifier, then the process that picked it becomes the leader. If not, another round follows.

Figure 3 shows a global program for k=3; it crucially relies on the new theory's decentralised decision making. We write $r.[x_1, \ldots, x_n] := [e_1, \ldots, e_n]$ to abbreviate $r.x_1 := e_1 \,;\, \cdots \,;\, r.x_n := e_n$, while we write $p.e \twoheadrightarrow [q_1.x_1, \ldots, q_n.x_n]$ to abbreviate $p.e \twoheadrightarrow q_1.x_1 \,;\, \cdots \,;\, p.e \twoheadrightarrow q_n.x_n$. First, the processes initialise five variables (lines 1–3): `seed` is used to pick random identifiers; `id1`, `id2`, and `id3` are used to store and compare identifiers; `leader` indicates whether or not the process was elected. Next, the processes enter the loop (lines 4–7), each of whose iterations represents one round: in every iteration, every process increments its seed, picks a random identifier, and shares it. When the maximal identifier is unique, the processes exit the loop. One process marks itself as leader (lines 8–10).

The point of this example is that the probabilistic version of Peleg's algorithm for cliques—actually, *any* leader election algorithm—*cannot* faithfully be implemented using centralised decision making. The reason is that centralised decision

```
1.  (P1.[seed, id1, id2, id3, leader] := [-1, -1, -1, -1, false] ||
2.   P2.[seed, id1, id2, id3, leader] := [-1, -1, -1, -1, false] ||
3.   P3.[seed, id1, id2, id3, leader] := [-1, -1, -1, -1, false]) ;
4.  while ⋀{r.!maxIsUnique(id1,id2,id3)}_{r∈{P1,P2,P3}}
5.      (P1.seed := seed+1 ; P1.id1 := random1(seed) ; P1.id1 ↠ [P3.id1, P2.id1] ||
6.       P2.seed := seed+1 ; P2.id2 := random2(seed) ; P2.id2 ↠ [P1.id2, P3.id2] ||
7.       P3.seed := seed+1 ; P3.id3 := random3(seed) ; P3.id3 ↠ [P2.id3, P1.id3]) ;
8.  if ⋀{r.id1 == max(id1,id2,id3)}_{r∈{P1,P2,P3}} (P1.leader := true) (skip) ;
9.  if ⋀{r.id2 == max(id1,id2,id3)}_{r∈{P1,P2,P3}} (P2.leader := true) (skip) ;
10. if ⋀{r.id3 == max(id1,id2,id3)}_{r∈{P1,P2,P3}} (P3.leader := true) (skip)
```

Fig. 3: Global program for probabilistic leader election in anonymous clique networks (k=3), using decentralised decision making

making inherently requires the presence of a distinguished process (to evaluate a one-party condition and share the outcome). However, the motivation to run a leader election algorithm in the first place is that such a distinguished process is not yet agreed upon. That is, centralised decision making requires *asymmetry* of processes, whereas leader election algorithms require *symmetry*. □

3 Setting the Stage: Data and Conditions

The topic of interest in this paper is "processes that communicate", rather than "data that are communicated". For this reason, we assume that there exists some underlying calculus of data (Sect. 3.1), but we omit most of its details; they are orthogonal to this paper's contributions. On top of it, we adopt a logic to write preconditions, postconditions, and conditions in if/while-statements (Sect. 3.2).

3.1 Data

Let $\mathbb{R} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \ldots\}$ denote a universe of *roles*, ranged over by p, q, r. Let $\mathbb{X} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$ denote a universe of *variables*, ranged over by x, y, z. Let $\mathbb{V} = \{\mathtt{error}, \mathtt{true}, \mathtt{false}, 0, 1, 2, \ldots\}$ denote a universe of *values*, ranged over by v (i.e., $\mathbb{V}$ contains at least a distinguished value `error`, booleans, and numbers, but we also use other data types in examples, including functions). Let $\mathbb{E}$ denote a universe of *expressions*, ranged over by e; it is induced by the following grammar:

$$e ::= \underbrace{r.x}_{\text{role-qualified variable}} \mid v \mid \underbrace{e_1\texttt{==}e_2 \mid e_1\texttt{<}e_2 \mid e_1\texttt{\&\&}e_2 \mid \texttt{!}e \mid e_1\texttt{+}e_2 \mid \cdots}_{\text{compound expressions}}$$

Let $\mathbb{S} = \mathbb{R} \rightharpoonup (\mathbb{X} \rightharpoonup \mathbb{V})$ denote a universe of *states* (i.e., partial functions from roles to partial functions from variables to values), ranged over by $\mathcal{S}$; the idea is that every state has a separate section for every role of interest, to model disjoint memory spaces. Let $\mathsf{eval} : \mathbb{S} \times \mathbb{E} \to \mathbb{V}$ denote a total *evaluation function*. For instance, $\mathsf{eval}_{\{\mathsf{A}\mapsto\{\mathtt{x}\mapsto 5,\mathtt{y}\mapsto 6\}\}}(\mathsf{A.x+A.y}) = 11$. We assume that bogus expressions are evaluated to `error`. For instance, $\mathsf{eval}_{\emptyset}(\mathtt{1+true}) = \mathtt{error}$.

Regarding terminology, we say that every role-qualified variable $r.x$ is "local to r". If every role-qualified variable that occurs in e is local to r, then e is "local to r". Regarding notation, if e is local to r, then we often move all "r."-qualifiers that occur in e to the front. For instance, we write `A.x+y` instead of `A.x+A.y`.

3.2 Conditions

We adopt the following basic logic over expressions in $\mathbb{E}$. Let $\mathbf{\Psi}$ denote a universe of *formulas*, ranged over by ϕ, χ, ψ; it is induced by the following grammar:

$$\phi, \chi, \psi \ ::= \ e \ \mid \ \neg\psi \ \mid \ \psi_1 \wedge \psi_2 \ \mid \ \forall\psi$$

Informally, given state $\mathcal{S}$, formulas have the following meaning relative to $\mathcal{S}$:

- Formula e is an **atom**: it is true in $\mathcal{S}$ iff e evaluates to `true` using $\mathcal{S}$.
- Formulas $\neg\psi$ and $\psi_1 \wedge \psi_2$ are a **negation** and a **conjunction**, as usual. (Negation and conjunction appear also at the level of formulas, and not just at the level of expressions, for technical convenience later on in this paper.)
- Formula $\forall\psi$ is a **tautology**: it is true in $\mathcal{S}$ iff ψ is true in every state.

Formally, an *interpretation function* maps formulas to the sets of states in which they are true, denoted by $[\![\text{-}]\!]$; it is induced by the following equations:

$$[\![e]\!] = \{\mathcal{S} \mid \mathsf{eval}_{\mathcal{S}}(e) = \texttt{true}\} \qquad [\![\neg\psi]\!] = \mathbb{S} \setminus [\![\psi]\!] \qquad [\![\psi_1 \wedge \psi_2]\!] = [\![\psi_1]\!] \cap [\![\psi_2]\!] \qquad [\![\forall\psi]\!] = \begin{cases} \mathbb{S} & \text{if: } [\![\psi]\!] = \mathbb{S} \\ \emptyset & \text{otherwise} \end{cases}$$

Regarding terminology, if every expression that occurs in ψ is local to r, then ψ is "local to r"; if so, the truth of ψ can be checked at r. Regarding notation, we often write $\bigwedge\{\psi_r\}_{r\in\{r_1,\ldots,r_n\}}$ instead of $\psi_{r_1} \wedge \cdots \wedge \psi_{r_n}$ if ψ_r is local to r for every $r \in \{r_1, \ldots, r_n\}$. Furthermore, we write $\psi_1 \vee \psi_2$ and $\psi_1 \rightarrow \psi_2$ for disjunction and implication. Finally, we write $\psi_1 \equiv \psi_2$ instead of $[\![\psi_1]\!] = [\![\psi_2]\!]$.

4 Global Programs: Base Calculus

To gently introduce the main components of the new theory, in this section, we present a base calculus of global programs, *without* if/while statements, but *with* a main theorem that covers both deadlock freedom and functional correctness.

Initially, we present the syntax and semantics (Sect. 4.1); subsequently, we present a predicate transformer (Sect. 4.2); finally, we present the main theorem, which relies on the predicate transformer (Sect. 4.3). In the next sections, we extend the base calculus to support decentralised decision making.

4.1 Syntax and Semantics

Let $\mathbf{\Gamma}$ and $\mathbb{G}$ denote universes of *global actions* and *global programs*, ranged over by γ and G; they are induced by the following grammar:

$$\gamma ::= q.y{:=}e \mid p.e \twoheadrightarrow q.y \qquad G ::= \mathbf{skip} \mid \gamma \mid G_1 \,;\, G_2 \mid G_1 \parallel G_2$$

Informally, these grammar elements have the following meaning:

- Global action $q.y{:=}e$ models an **assignment** of the value of expression e to variable y at role q. As an extra constraint, e is local to q. Regarding notation, we often omit "q."-qualifiers from e. For instance, we write $\mathsf{A.z{:=}x{+}y}$ instead of $\mathsf{A.z{:=}A.x{+}A.y}$. Also, we write $\mathsf{eval}_{\mathcal{S}}(q.y{:=}e)$ instead of $q.y{:=}\mathsf{eval}_{\mathcal{S}}(e)$.
- Global action $p.e \twoheadrightarrow q.y$ models a synchronous **communication** of the value of expression e at role p into variable y at role q. As extra constraints, e is local to p, and $p \neq q$. Regarding notation, we often omit "p."-qualifiers from e. Also, we write $\mathsf{eval}_{\mathcal{S}}(p.e \twoheadrightarrow q.y)$ instead of $p.\mathsf{eval}_{\mathcal{S}}(e) \twoheadrightarrow q.y$.
- Global program **skip** prescribes an **empty execution**.
- Global program $G_1 \,;\, G_2$ prescribes a ***weak*** **sequence** of G_1 and G_2. The idea is that it resembles a conventional *strong* sequence (i.e., in-order execution), except that it also allows global actions in G_2 *that are independent of those in* G_1 to be executed already before G_1 is done (i.e., out-of-order).
  For instance, in $\mathsf{A.x{:=}5} \,;\, \mathsf{B.y{:=}6}$, the assignment at Bob is independent of the assignment at Alice, so they may be executed out-of-order. In contrast, in $\mathsf{A.x{:=}5} \,;\, \mathsf{A.x{+}1} \twoheadrightarrow \mathsf{B.y}$, the communication from Alice to Bob depends on the assignment at Alice, so they must be executed in-order. In general, when two global actions have disjoint *subjects* (i.e., participating roles), they are considered independent and may be executed out-of-order.
  Out-of-order execution of global actions with disjoint subjects is common in choreographic programming: it was first introduced by Carbone and Montesi to deal with latent concurrency among roles in global action sequences [10].
- Global program $G_1 \parallel G_2$ prescribes an **interleaving** of G_1 and G_2.

Formally, we define the operational semantics of global programs at two "layers".

(1) The "top layer" consists of an *abstract termination relation*, denoted by $\downarrow$, and an *abstract labelled reduction relation*, denoted by $\rightarrow$ in the style of process algebra (e.g., [2]). More precisely, $G\downarrow$ means that G can terminate, while $G \xrightarrow{\psi,\gamma} G'$ means that G can reduce to G' when ψ is true (i.e., conditionally) by executing γ. For instance, the following *abstract execution* is possible:

$$\mathsf{A.x{:=}5} \,;\, \mathsf{A.x{+}1} \twoheadrightarrow \mathsf{B.y} \xrightarrow{\mathtt{true},\mathsf{A.x:=5}} \mathbf{skip} \,;\, \mathsf{A.x{+}1} \twoheadrightarrow \mathsf{B.y} \xrightarrow{\mathtt{true},\mathsf{A.x{+}1}\twoheadrightarrow\mathsf{B.y}} \mathbf{skip} \,;\, \mathbf{skip}\downarrow$$

First, the global program reduces by executing an assignment; next, it reduces by executing a communication; next, it terminates. For simplicity, **skip**s are not automatically cleaned up by the reduction rules (but they could be).

Relations $\downarrow$ and $\rightarrow$ are induced by the rules in Fig. 4a. Most rules are standard [2]. Notably, in this section, every reduction is *unconditional* (i.e., labelled with `true`) due to rule [$\rightarrow$-Act]. The only special rule is rule [$\rightarrow$-Seq2]: it states that if G_2 can reduce to G_2' by executing γ (right premise), and if γ is independent of G_1 (left premise), then $G_1 \,;\, G_2$ can reduce accordingly (conclusion). We note that independence is defined in terms of disjointness of subjects, as explained above. For instance, the following abstract out-of-order execution is possible:

$$\frac{}{\mathbf{skip}\downarrow}\ [\downarrow\text{-Skip}] \qquad \frac{G_1\downarrow \quad G_2\downarrow}{G_1\,;\,G_2\downarrow}\ [\downarrow\text{-Seq}] \qquad \frac{G_1\downarrow \quad G_2\downarrow}{G_1 \parallel G_2\downarrow}\ [\downarrow\text{-Par}] \qquad \frac{\psi = \mathtt{true}}{\gamma \xrightarrow{\psi,\gamma} \mathbf{skip}}\ [\rightarrow\text{-Act}]$$

$$\frac{G_1 \xrightarrow{\psi,\gamma} G_1'}{G_1\,;\,G_2 \xrightarrow{\psi,\gamma} G_1'\,;\,G_2}\ [\rightarrow\text{-Seq1}] \qquad \frac{\mathsf{subj}(G_1) \cap \mathsf{subj}(\gamma) = \emptyset \quad G_2 \xrightarrow{\psi,\gamma} G_2'}{G_1\,;\,G_2 \xrightarrow{\psi,\gamma} G_1\,;\,G_2'}\ [\rightarrow\text{-Seq2}]$$

$$\frac{G_1 \xrightarrow{\psi,\gamma} G_1'}{G_1 \parallel G_2 \xrightarrow{\psi,\gamma} G_1' \parallel G_2}\ [\rightarrow\text{-Par1}] \qquad \frac{G_2 \xrightarrow{\psi,\gamma} G_2'}{G_1 \parallel G_2 \xrightarrow{\psi,\gamma} G_1 \parallel G_2'}\ [\rightarrow\text{-Par2}]$$

(a) Base calculus

$$\frac{\psi = \bigwedge\{e_r\}_{r\in R} \quad \gamma = 1^R}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1\ G_2 \xrightarrow{\psi,\gamma} G_1}\ [\rightarrow\text{-If1}] \qquad \frac{\psi = \bigwedge\{\neg e_r\}_{r\in R} \quad \gamma = 2^R}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1\ G_2 \xrightarrow{\psi,\gamma} G_2}\ [\rightarrow\text{-If2}]$$

$$\frac{\psi = \bigwedge\{e_r\}_{r\in R} \quad \gamma = 1^R}{\mathbf{while}\ \bigwedge\{e_r\}_{r\in R}\ \{\psi_{\mathrm{inv}}\}\ G \xrightarrow{\psi,\gamma} G\,;\,\mathbf{while}\ \bigwedge\{e_r\}_{r\in R}\ \{\psi_{\mathrm{inv}}\}\ G}\ [\rightarrow\text{-While1}]$$

$$\frac{\psi = \bigwedge\{\neg e_r\}_{r\in R} \quad \gamma = 2^R}{\mathbf{while}\ \bigwedge\{e_r\}_{r\in R}\ \{\psi_{\mathrm{inv}}\}\ G \xrightarrow{\psi,\gamma} \mathbf{skip}}\ [\rightarrow\text{-While2}]$$

(b) Extension with if/while-statements – *explained in Sect. 5*

$$\frac{R = R_1 \cup R_2 \quad R_1 \neq \emptyset \text{ implies } G_1\downarrow \quad R_2 \neq \emptyset \text{ implies } G_2\downarrow}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2|_{R_2}\downarrow}\ [\downarrow\text{-NIf}]$$

$$\frac{r \in R \setminus (R_1 \cup R_2) \quad \psi = e_r \quad \gamma = 1^{\{r\}}}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2|_{R_2} \xrightarrow{\psi,\gamma} \mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1\cup\{r\}}\ G_2|_{R_2}}\ [\rightarrow\text{-NIf1}]$$

$$\frac{r \in R \setminus (R_1 \cup R_2) \quad \psi = \neg e_r \quad \gamma = 2^{\{r\}}}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2|_{R_2} \xrightarrow{\psi,\gamma} \mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2|_{R_2\cup\{r\}}}\ [\rightarrow\text{-NIf2}]$$

$$\frac{G_1 \xrightarrow{\psi,\gamma} G_1' \quad \mathsf{subj}(\gamma) \subseteq R_1 \setminus R_2}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2|_{R_2} \xrightarrow{\psi,\gamma} \mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1'|_{R_1}\ G_2|_{R_2}}\ [\rightarrow\text{-NIf3}]$$

$$\frac{G_2 \xrightarrow{\psi,\gamma} G_2' \quad \mathsf{subj}(\gamma) \subseteq R_2 \setminus R_1}{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2|_{R_2} \xrightarrow{\psi,\gamma} \mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ G_1|_{R_1}\ G_2'|_{R_2}}\ [\rightarrow\text{-NIf4}]$$

$$\frac{\mathbf{if}\ \bigwedge\{e_r\}_{r\in R}\ (G\,;\,\mathbf{while}\ \bigwedge\{e_r\}_{r\in R}\ \{\psi_{\mathrm{inv}}\}\ G|_\emptyset)|_\emptyset\ \mathbf{skip}|_\emptyset \xrightarrow{\psi,\gamma} G'}{\mathbf{while}\ \bigwedge\{e_r\}_{r\in R}\ \{\psi_{\mathrm{inv}}\}\ G|_\emptyset \xrightarrow{\psi,\gamma} G'}\ [\rightarrow\text{-NWhile}]$$

(c) Extension with non-blocking if/while-statements – *explained in Sect. 6*

Fig. 4: Abstract operational semantics of global programs ("top layer")

$$\frac{G\downarrow}{(G,\mathcal{S})\downarrow}\,[\downarrow\downarrow] \qquad \frac{G \xrightarrow{\psi,\gamma} G' \quad \mathcal{S} \in [\![\psi]\!] \quad \gamma^{\mathrm{c}} = \mathsf{eval}_{\mathcal{S}}(\gamma)}{(G,\mathcal{S}) \xrightarrow{\gamma^{\mathrm{c}}} (G', \mathsf{effect}(\gamma^{\mathrm{c}}, \mathcal{S}))}\,[\rightarrow] \qquad \begin{array}{l}\mathsf{effect}(q.y{:=}v, \mathcal{S}) = \mathcal{S}[v/q.y] \\ \mathsf{effect}(p.v \twoheadrightarrow q.y, \mathcal{S}) = \mathcal{S}[v/q.y]\end{array}$$

$$\mathcal{S}[v/q.y] = \{r \mapsto \mathcal{S}(r) \mid q \neq r\} \cup \{q \mapsto \{x \mapsto \mathcal{S}(q)(x) \mid x \neq y\} \cup \{y \mapsto v\}\}$$

Fig. 5: Concrete operational semantics of global programs ("bottom layer")

$$\mathsf{A.x{:=}5}\,;\mathsf{B.y{:=}6} \xrightarrow{\mathtt{true},\mathsf{B.y:=6}} \mathsf{A.x{:=}5}\,;\mathbf{skip} \xrightarrow{\mathtt{true},\mathsf{A.x:=5}} \mathbf{skip}\,;\mathbf{skip}\downarrow$$

(2) The "bottom layer" consists of a *concrete termination predicate*, denoted by $\downarrow$ (same symbol as before), and a *concrete labelled reduction relation*, denoted by $\rightarrow$ (ditto). The idea is that the bottom layer enriches the top layer by taking into account states, in terms of *configurations* of the form $(G, \mathcal{S})$. More precisely, $(G,\mathcal{S})\downarrow$ means that G can terminate in $\mathcal{S}$, while $(G,\mathcal{S}) \xrightarrow{\gamma^{\mathrm{c}}} (G',\mathcal{S}')$ means that G can reduce to G' by executing γ^{c} in $\mathcal{S}$ to obtain $\mathcal{S}'$. We write γ^{c}—with a superscript "c"—to indicate that it is a "concrete" global action in which every expression has been evaluated to a value (using $\mathcal{S}$). For instance, the following *concrete execution* is possible:

$$\begin{array}{r} (\mathsf{A.x{:=}5}\,;\mathsf{A.x{+}1} \twoheadrightarrow \mathsf{B.y}, \{\mathsf{A} \mapsto \{\mathtt{x} \mapsto 0\}, \mathsf{B} \mapsto \{\mathtt{y} \mapsto 0\}\}) \\ \xrightarrow{\mathsf{A.x:=5}} \quad (\mathbf{skip}\,;\mathsf{A.x{+}1} \twoheadrightarrow \mathsf{B.y}, \{\mathsf{A} \mapsto \{\mathtt{x} \mapsto 5\}, \mathsf{B} \mapsto \{\mathtt{y} \mapsto 0\}\}) \\ \xrightarrow{\mathsf{A.6} \twoheadrightarrow \mathsf{B.y}} \quad (\mathbf{skip}\,;\mathbf{skip}, \{\mathsf{A} \mapsto \{\mathtt{x} \mapsto 5\}, \mathsf{B} \mapsto \{\mathtt{y} \mapsto 6\}\})\downarrow \end{array}$$

Relations $\downarrow$ and $\rightarrow$ are induced by the rules in Fig. 5. Rule $[\downarrow\downarrow]$ states that if G can terminate, then so can $(G,\mathcal{S})$, regardless of $\mathcal{S}$. More interestingly, rule $[\rightarrow]$ states that if G can reduce to G' when ψ is true by executing γ (left premise), and if ψ is indeed true in $\mathcal{S}$ (middle premise), and if γ^{c} is the "concretisation" of γ such that every expression is first evaluated using $\mathcal{S}$ (right premise), then $(G,\mathcal{S})$ can reduce accordingly, and the *effect* of γ^{c} is applied to $\mathcal{S}$ (conclusion); the latter means that a variable is bound to a new value in $\mathcal{S}$, formalised using "substitution notation". For instance (cf. second reduction in the concrete execution above), if $\mathcal{S} = \{\mathsf{A} \mapsto \{\mathtt{x} \mapsto 5\}, \mathsf{B} \mapsto \{\mathtt{y} \mapsto 0\}\}$, then $\mathsf{effect}(\mathsf{eval}_{\mathcal{S}}(\mathsf{A.x{+}1} \twoheadrightarrow \mathsf{B.y}), \mathcal{S}) = \mathsf{effect}(\mathsf{A.6} \twoheadrightarrow \mathsf{B.y}, \mathcal{S}) = \{\mathsf{A} \mapsto \{\mathtt{x} \mapsto 5\}, \mathsf{B} \mapsto \{\mathtt{y} \mapsto 6\}\}$.

Our formalisation of the operational semantics has two novelties:

- *Two-layered approach* – In existing work on stateful choreographic programming (e.g., [14,19]), abstract and concrete operational semantics are merged into one. An advantage of keeping them separate is that it enables us to prove the main theorems also in a layered fashion; this simplifies our proofs.
- *Semantic reordering* – In existing work on choreographic programming (e.g., [10,41]), weak sequencing is formalised using a structural congruence relation in the style of pi-calculus (e.g., [49]), including special "swap rules" to syntactically reorder independent global actions. In contrast, rule [→-Seq2] semantically reorders them; this simplifies our proofs. Our approach, inspired by Rensink and Wehrheim [48], essentially generalises the formalisation of asynchronous action prefixing in multiparty session types [24].

$$\frac{R \neq \emptyset}{\checkmark_R(\mathbf{skip})}\ [\checkmark\text{-Skip}] \qquad \frac{q \in R}{\checkmark_R(q.y := e)}\ [\checkmark\text{-Act1}] \qquad \frac{p, q \in R}{\checkmark_R(p.e \twoheadrightarrow q.y)}\ [\checkmark\text{-Act2}]$$

$$\frac{\checkmark_R(G_1) \quad \checkmark_R(G_2)}{\checkmark_R(G_1\,;\,G_2)}\ [\checkmark\text{-Seq}] \qquad \frac{\checkmark_R(G_1) \quad \checkmark_R(G_2) \quad \mathsf{chan}(G_1) \cap \mathsf{chan}(G_2) = \emptyset}{\checkmark_R(G_1 \parallel G_2)}\ [\checkmark\text{-Par}]$$

(a) Base calculus

$$\frac{\checkmark_R(G_1) \quad \checkmark_R(G_2)}{\checkmark_R(\mathbf{if}\ \bigwedge\{e_r\}_{r \in R}\ G_1\ G_2)}\ [\checkmark\text{-If}] \qquad \frac{\checkmark_R(G)}{\checkmark_R(\mathbf{while}\ \bigwedge\{e_r\}_{r \in R}\ \{\psi_{\text{inv}}\}\ G)}\ [\checkmark\text{-While}]$$

(b) Extension with if/while-statements – *explained in Sect. 5*

$$\frac{\begin{array}{l}\checkmark_R(G_1) \quad \checkmark_R(G_2) \quad R_1, R_2 \subseteq R \\ R_1 \neq \emptyset \text{ implies } R_2 = \emptyset \\ R_2 \neq \emptyset \text{ implies } R_1 = \emptyset\end{array}}{\checkmark_R(\mathbf{if}\ \bigwedge\{e_r\}_{r \in R}\ G_1|_{R_1}\ G_2|_{R_2})}\ [\checkmark\text{-NIf}] \qquad \frac{\checkmark_R(G)}{\checkmark_R(\mathbf{while}\ \bigwedge\{e_r\}_{r \in R}\ \{\psi_{\text{inv}}\}\ G|_{\emptyset})}\ [\checkmark\text{-NWhile}]$$

(c) Extension with non-blocking if/while-statements – *explained in Sect. 6*

Fig. 6: Well-formedness of global programs

We end this subsection with a *well-formedness relation*, denoted by ✓, to check a few basic syntactic properties of global programs; it is induced by the rules in Fig. 6a. For now, there are two aims (to be extended in subsequent sections for if/while-statements):

1. Rules [✓-Act1] and [✓-Act2] ensure that R contains all roles that occur in G. The idea is that when we project G onto every role in R (Sect. 7), we get a local program for every remaining subject of G (i.e., when G is the remaining global program, R may contain roles that participated in the past, but no longer in the future). Thus, R spans the whole distributed system.

2. Rule [✓-Par] ensures that the *channels* (i.e., sender–receiver pairs) that occur in G_1 and G_2 are disjoint; this is a common assumption in choreographic programming (e.g., [8]). The idea is that when a communication happens in $G_1 \parallel G_2$, it must be unambiguously clear whether it happened in G_1 or in G_2; otherwise, the operational equivalence theorem cannot be proved (Sect. 7).

4.2 Predicate Transformer

In the next subsection, the main theorem for global programs will be as follows (informally): if the global program is well-formed, and if the precondition is true in the initial state, then deadlock freedom and functional correctness are provided. In this subsection, we present a technique to automatically compute preconditions such that the main theorem can indeed be formulated and proved.

$$
\begin{aligned}
\phi(\textbf{skip}, \chi) &= \chi \\
\phi(q.y := e, \chi) &= \chi[e/q.y] \\
\phi(p.e \twoheadrightarrow q.y, \chi) &= \chi[e/q.y] \\
\phi(G_1\,;\, G_2, \chi) &= \phi(G_1, \phi(G_2, \chi)) \\
\phi(G_1 \parallel G_2, \chi) &= \begin{cases} \phi(G_1, \phi(G_2, \chi)) & \\ \quad \text{if: } G_1 \# G_2 & \\ \texttt{false} & \\ \quad \text{otherwise} & \end{cases}
\end{aligned}
$$

(a) Base calculus

$$
\begin{aligned}
&\phi(\textbf{if } \textstyle\bigwedge\{e_r\}_{r\in R}\, G_1\, G_2, \chi) = \\
&\quad (\textstyle\bigwedge\{\ \ e_r\}_{r\in R} \to \phi(G_1, \chi)) \wedge \\
&\quad (\textstyle\bigwedge\{\neg e_r\}_{r\in R} \to \phi(G_2, \chi)) \wedge \\
&\quad (\textstyle\bigwedge\{e_{r_1} \to e_{r_2}\}_{r_1, r_2 \in R}) \\
&\phi(\textbf{while } \textstyle\bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G, \chi) = \\
&\quad \psi_{\text{inv}} \wedge \forall(\psi_{\text{inv}} \to (\\
&\qquad (\textstyle\bigwedge\{\ \ e_r\}_{r\in R} \to \phi(G, \psi_{\text{inv}})) \wedge \\
&\qquad (\textstyle\bigwedge\{\neg e_r\}_{r\in R} \to \chi) \wedge \\
&\qquad (\textstyle\bigwedge\{e_{r_1} \to e_{r_2}\}_{r_1, r_2 \in R})))
\end{aligned}
$$

(b) Extension with if/while-statements – *explained in Sect. 5*

$$
\begin{aligned}
&\phi(\textbf{if } \textstyle\bigwedge\{e_r\}_{r\in R}\, G_1|_{R_1}\, G_2|_{R_2}, \chi) = \begin{cases} \phi(\textbf{if } \bigwedge\{e_r\}_{r\in R}\, G_1\, G_2, \chi) & \text{if: } R_1 = \emptyset = R_2 \\ \phi(G_2, \chi) \wedge \bigwedge\{\neg e_r\}_{r\in R\setminus R_2} & \text{if: } R_1 = \emptyset \neq R_2 \\ \phi(G_1, \chi) \wedge \bigwedge\{\ \ e_r\}_{r\in R\setminus R_1} & \text{if: } R_1 \neq \emptyset = R_2 \\ \texttt{false} & \text{if: } R_1 \neq \emptyset \neq R_2 \end{cases} \\
&\phi(\textbf{while } \textstyle\bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G|_{\emptyset}, \chi) = \phi(\textbf{while } \textstyle\bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G, \chi)
\end{aligned}
$$

(c) Extension with non-blocking if/while-statements – *explained in Sect. 6*

Fig. 7: Predicate transformer to compute preconditions

Let ϕ denote a *predicate transformer function*; it is defined by the equations in Fig. 7a, where $\chi[e/q.y]$ denotes substitution of e for $q.y$ in χ. In words, ϕ consumes a global program G and a postcondition χ as input, and it produces a precondition $\phi(G, \chi)$ as output. The idea is that ϕ is *sound*: if $\phi(G, \chi)$ is true in the initial state, then after executing G, χ is true in the final state. Essentially, Fig. 7a is an adaptation of Dijkstra's predicate transformer to compute *weakest preconditions* for sequential code [26], denoted by `wp`. More precisely:

- For $q.y := e$, the definitions of ϕ and `wp` are the same; for $p.e \twoheadrightarrow q.y$ (absent in Dijkstra's work), ϕ works similarly. Figure 8a shows an example: if `A.x` is `5` (computed precondition), then after the communication of `A.x+1` at Alice into `B.y` at Bob (global program), the sum of `A.x` and `B.y` is `11` (postcondition). We note that the postcondition relates variables *at different roles*; this is straightforwardly supported by ϕ, without extra manual effort.
- For $G_1\,;\, G_2$, the definitions of ϕ and `wp` are the same as well: first, χ is used as a postcondition of G_2 to compute a precondition $\phi(G_2, \chi)$; next, $\phi(G_2, \chi)$ is used as a postcondition of G_1 to compute a precondition $\phi(G_1, \phi(G_2, \chi))$. Such a "backwards" computation of a precondition corresponds to the "forwards" execution of the sequence: initially, $\phi(G_1, \phi(G_2, \chi))$ is true; subsequently, $\phi(G_2, \chi)$ is true after executing G_1; finally, χ is true after executing

$$\begin{aligned}&\phi(\texttt{A.x+1} \twoheadrightarrow \texttt{B.y}, \texttt{A.x+B.y==11})\\ &= \texttt{A.x+A.x+1==11}\\ &\equiv \texttt{A.x+A.x==10} \equiv \texttt{A.x==5}\end{aligned}$$

(a) Communication

$$\begin{aligned}&\phi(\gamma\,;\texttt{A.x+1} \twoheadrightarrow \texttt{B.y}, \texttt{A.x+B.y==11})\\ &= \phi(\gamma, \phi(\texttt{A.x+1} \twoheadrightarrow \texttt{B.y}, \texttt{A.x+B.y==11}))\\ &= \phi(\gamma, \texttt{A.x+A.x+1==11}) = \texttt{5+5+1==11} \equiv \texttt{true}\end{aligned}$$

(b) Sequence

$$\begin{aligned}&\phi(\gamma\,;\textbf{if } (\texttt{A.x==5} \wedge \texttt{B.y==6})\ \texttt{B.y:=7}\ \textbf{skip}, \chi)\\ &= \phi(\gamma, \phi(\textbf{if } (\texttt{A.x==5} \wedge \texttt{B.y==6})\ \texttt{B.y:=7}\ \textbf{skip}, \chi))\\ &= \phi(\gamma, (\texttt{A.x==5} \wedge \texttt{B.y==6} \rightarrow \phi_1) \wedge (\neg\texttt{A.x==5} \wedge \neg\texttt{B.y==6} \rightarrow \phi_2) \wedge (\texttt{A.x==5} \leftrightarrow \texttt{B.y==6}))\\ &= (\texttt{5==5} \wedge \texttt{B.y==6} \rightarrow \phi_1[\texttt{5/A.x}]) \wedge (\neg\texttt{5==5} \wedge \neg\texttt{B.y==6} \rightarrow \phi_2[\texttt{5/A.x}]) \wedge (\texttt{5==5} \leftrightarrow \texttt{B.y==6})\\ &\equiv (\texttt{B.y==6} \rightarrow \phi_1[\texttt{5/A.x}]) \wedge (\texttt{false} \rightarrow \phi_2[\texttt{5/A.x}]) \wedge \texttt{B.y==6} \equiv \phi_1[\texttt{5/A.x}] \wedge \texttt{B.y==6}\end{aligned}$$

(c) Conditional choice – *explained in Sect. 5*. Let $\phi_1 = \phi(\texttt{B.y:=7}, \chi)$, $\phi_2 = \phi(\textbf{skip}, \chi)$.

Fig. 8: Examples of ϕ. Let $\gamma = \texttt{A.x:=5}$.

G_2. Figure 8b shows an example: if `true` is true (i.e., unconditionally), after executing the global program, the sum of `A.x` and `B.y` is `11`.
However, unlike Dijkstra's setting (i.e., strong sequencing), there is a caveat in our setting (i.e., weak): G_1 and G_2 may be executed out-of-order. This makes proving the soundness of ϕ more challenging than in Dijkstra's work (notably: establishing the correspondence between backwards computation of a precondition and forwards execution of the sequence).

– For $G_1 \parallel G_2$ (absent in Dijkstra's work), the definition of ϕ is inspired by the notion of *disjoint parallelism* in Hoare logic [33,1]. There are two cases. If G_1 and G_2 are *non-interfering*, which means that the variables that occur in G_1 and G_2 are disjoint, denoted as $G_1 \# G_2$, then the order in which G_1 and G_2 are executed does not affect the truth/falsehood of the postcondition; in that case, a precondition is computed by assuming, arbitrarily, in-order execution of G_1 and G_2 (but any other interleaving would work as well). If G_1 and G_2 are interfering, then ϕ yields `false`, so no state satisfies the precondition. This is sound but not complete (i.e., there exist deadlock-free and functionally-correct global programs for which the computed precondition is nevertheless `false`). For our present purposes, however, ϕ is "complete enough" (e.g., all examples in Sect. 2 and Appx. A are supported).[4]

The following proposition follows almost directly from the definitions. It states that if $\phi(\gamma, \chi)$ is true in $\mathcal{S}$, then χ is true in $\mathcal{S}'$, after executing γ.

Proposition 1. *If* $\mathcal{S} \in [\![\phi(\gamma, \chi)]\!]$ *and* $\mathcal{S}' = \mathsf{effect}(\mathsf{eval}_{\mathcal{S}}(\gamma), \mathcal{S})$, *then* $\mathcal{S}' \in [\![\chi]\!]$.

[4] Even though ϕ requires non-interference, interleaving ($\parallel$) offers additional expressive power beyond weak sequencing (;). This is because non-interference (for $\parallel$) is defined in terms of disjointness *of variables*, whereas independence (for ;) is defined in terms of disjointness *of roles*. For instance, `A.x:=5` and `A.y:=6` are non-interfering, but not independent. Consequently, `A.x:=5` $\parallel$ `A.y:=6` allows the assignments to happen in any order, whereas `A.x:=5` ; `A.y:=6` requires them to happen from left to right.

4.3 Deadlock Freedom and Functional Correctness

The aim of this subsection is to formulate and prove the main theorem for global programs, which covers both deadlock freedom and functional correctness.

To give a uniform presentation across Sects. 4–6, we formulate the lemmas and theorem for the base calculus in this section in a way that they are reusable—*verbatim*—for the extensions in the next sections. As a result, some formulations are more restrictive than necessary for the base calculus, but this is fine.

The first two lemmas pertain to ϕ's soundness. The first lemma states that if G is well-formed and can terminate, then the truth of $\phi(G, \chi)$ implies the truth of χ (i.e., the postcondition *has been* brought about). The second lemma states that if G is well-formed and can reduce to G' when ψ is true by executing γ, then the truth of $\phi(G, \chi) \wedge \psi$ implies the truth of χ, after executing $\gamma\,;G'$ (i.e., the postcondition *is being* brought about by executing γ).

Lemma 1. *If* $\checkmark_R(G)$ *and* $G{\downarrow}$*, then* $[\![\phi(G,\chi)]\!] \subseteq [\![\chi]\!]$.

Proof. By induction on the derivation of $G{\downarrow}$. □

Lemma 2. *If* $\checkmark_R(G)$ *and* $G \xrightarrow{\psi,\gamma} G'$*, then* $[\![\phi(G,\chi) \wedge \psi]\!] \subseteq [\![\phi(\gamma\,;G',\chi)]\!]$.

Proof. By induction on the derivation of $G \xrightarrow{\psi,\gamma} G'$. The interesting case is rule [→-Seq2], with $G = G_1\,;G_2$. We need to prove the following inclusions:

$$[\![\phi(G_1,\phi(G_2,\chi)) \wedge \psi]\!] \subseteq [\![\phi(G_1\,;\gamma\,;G_2',\chi)]\!] \subseteq [\![\phi(\gamma\,;G_1\,;G_2',\chi)]\!]$$

The first inclusion can be proved using the induction hypothesis and $G_2 \xrightarrow{\psi,\gamma} G_2'$ (right premise of rule [→-Seq2]). The second inclusion can be proved using $\mathsf{subj}(G_1) \cap \mathsf{subj}(\gamma) = \emptyset$ (left premise) and $\checkmark_R(G)$, to establish that *the variables* that occur in G_1 and γ are disjoint as well (i.e., G_1 and γ are non-interfering). □

The next lemma states that well-formedness is preserved by reduction.

Lemma 3. *If* $\checkmark_R(G)$ *and* $[\![\phi(G,\chi)]\!] \neq \emptyset$ *and* $G \xrightarrow{\psi,\gamma} G'$*, then* $\checkmark_R(G')$.

Proof. By induction on the derivation of $G \xrightarrow{\psi,\gamma} G'$. □

The next lemma states that if G is well-formed, and if $\phi(G,\chi)$ is true in $\mathcal{S}$, then either G can terminate, or G can reduce to G' (i.e., G is not stuck).

Lemma 4. *If* $\checkmark_R(G)$ *and* $\mathcal{S} \in [\![\phi(G,\chi)]\!]$*, then either* $G{\downarrow}$*, or there exist* ψ,γ,G' *such that* $G \xrightarrow{\psi,\gamma} G'$ *and* $\mathcal{S} \in [\![\psi]\!]$.

Proof. By induction on the derivation of $\checkmark_R(G)$. □

Now, our main theorem for global programs states that if G is well-formed, and if $\phi(G,\chi)$ is true in $\mathcal{S}$, and if $(G,\mathcal{S})$ has a sequence of reductions to $(G^\dagger,\mathcal{S}^\dagger)$, then either $(G^\dagger,\mathcal{S}^\dagger)$ can terminate and χ is true in $\mathcal{S}^\dagger$, or $(G^\dagger,\mathcal{S}^\dagger)$ can reduce. Thus, an execution of $(G,\mathcal{S})$ consists of either finitely many reductions, followed by termination, or infinitely many (i.e., **deadlock freedom**); in the former case, upon termination, the postcondition is true (i.e., **functional correctness**).

Theorem 3. *If* $\checkmark_R(G)$ *and* $\mathcal{S} \in [\![\phi(G,\chi)]\!]$ *and* $(G,\mathcal{S}) \xrightarrow{\gamma_1^c} \cdots \xrightarrow{\gamma_n^c} (G^\dagger,\mathcal{S}^\dagger)$*, then:*

1. *Either* $(G^\dagger,\mathcal{S}^\dagger)\downarrow$*, or there exist* $\gamma^c, G^\ddagger, \mathcal{S}^\ddagger$ *such that* $(G^\dagger,\mathcal{S}^\dagger) \xrightarrow{\gamma^c} (G^\ddagger,\mathcal{S}^\ddagger)$.
2. *If* $(G^\dagger,\mathcal{S}^\dagger)\downarrow$*, then* $\mathcal{S}^\dagger \in [\![\chi]\!]$.

Proof. First, we inductively apply Prop. 1 and Lems. 2–3, along the reduction sequence to prove $\checkmark_R(G^\dagger)$ and $\mathcal{S}^\dagger \in [\![\phi(G^\dagger,\chi)]\!]$. Next, we apply Lem. 4 to prove deadlock freedom and Lem. 1 to prove functional correctness, using Fig. 5. □

5 Global Programs: If/While-Statements

In the previous section, to gently introduce the main components of our theory, we presented a base calculus of global programs. In this section, we extend it with if/while-statement to support decentralised decision making.

5.1 Syntax and Semantics

Recall that $\mathbf{\Gamma}$ and $\mathbb{G}$ denote universes of global actions and global programs, ranged over by γ and G; they are induced by the following extended grammar:

$$\begin{aligned} \gamma &::= \cdots \text{ (page 8)} \mid i^R \\ G &::= \cdots \text{ (page 8)} \mid \textbf{if } \textstyle\bigwedge\{e_r\}_{r\in R}\, G_1\, G_2 \mid \textbf{while } \textstyle\bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G \end{aligned}$$

Informally, the new grammar elements have the following meaning:

- Global action i^R, with $i \in \{1,2\}$, models a **collection of private decisions** at all roles in R *together* (i.e., at the same time). In case of an if-statement, i=1 and i=2 indicate entering the then-branch and else-branch; in case of a while-statement, i=1 and i=2 indicate (re)entering the loop and exiting it.
- Global program **if** $\bigwedge\{e_r\}_{r\in R}\, G_1\, G_2$ prescribes a **conditional choice** of G_1 and G_2. The idea is that every role $r \in R$ privately evaluates its own conjunct e_r of multiparty condition $\bigwedge\{e_r\}_{r\in R}$ and, based on the outcome, privately decides to enter G_1 or G_2. As a result, we have *three cases* to consider:
  - **Case A:** If e_r is true for every $r \in R$, then everyone enters G_1.
  - **Case B:** If e_r is false for every $r \in R$, then everyone enters G_2.
  - **Case C:** If e_{r_1} is true, but e_{r_2} is false, for some $r_1, r_2 \in R$, then someone enters G_1, but someone else enters G_2.

  Cases A and B are the "good" situations in which the roles are unanimous. In contrast, case C is the "bad" situation that leads to deadlock.
  For simplicity, in this section, we assume that roles make private decisions *together* (i.e., at the same time), using two *synchronisation barriers*. Intuitively, in operational terms, this means that for every role r: first, it privately evaluates its own conjunct e_r; next, it reaches one of two barriers, depending on the truth/falsehood of e_r; next, it waits until every other role has privately evaluated a conjunct and reached a barrier as well. In cases A and B,

all roles eventually reach the same barrier, so it breaks, and all roles enter one branch together; in case C, the roles never reach the same barrier—they are divided—so neither one of them breaks, and the roles get stuck.
(We note that barriers are often undesirable in distributed systems. In the next section, therefore, we also extend the base calculus with barrier-free if/while-statements. However, as the technical details of the barrier-free versions are considerably more complicated than the barrier-based versions, but partly rely on similar principles, we present the barrier-based ones first.)
An if-statement cannot terminate: a decision must be made.

– Global program **while** $\bigwedge\{e_r\}_{r\in R}\,\{\psi_{\text{inv}}\}\,G$ prescribes a **conditional loop** of G. The idea is similar to **if** $\bigwedge\{e_r\}_{r\in R}\,G_1\,G_2$, including non-termination. Condition ψ_{inv} is the *loop invariant*; it does not affect the operational semantics of while-statements, but it is used to compute preconditions.

Formally, for if/while-statements, $\rightarrow$ is induced by the rules in Fig. 4b (page 10). The *presence* of rules [→-If1] and [→-If2] corresponds to cases A and B, whereas the *absence* of other rules corresponds to case C (i.e., there are no reductions when roles are not unanimous). For instance, when $G = \mathsf{A.x}:=5\,;\,\textbf{if}\,(\mathsf{A.x{=}{=}5} \wedge \mathsf{B.y{=}{=}6})\,\mathsf{B.y}:=7\,\textbf{skip}$, the following two abstract executions are possible:

$$G \xrightarrow{\substack{\mathsf{true},\\ \mathsf{A.x}:=5}} \bullet \xrightarrow[1^{\{\mathsf{A},\mathsf{B}\}}]{\mathsf{A.x{=}{=}5}\wedge\mathsf{B.y{=}{=}6},} \bullet \xrightarrow{\substack{\mathsf{true},\\ \mathsf{B.y}:=7}} \bullet\!\downarrow \qquad G \xrightarrow{\substack{\mathsf{true},\\ \mathsf{A.x}:=5}} \bullet \xrightarrow[2^{\{\mathsf{A},\mathsf{B}\}}]{\neg\mathsf{A.x{=}{=}5}\wedge\neg\mathsf{B.y{=}{=}6},} \bullet\!\downarrow$$

First, G reduces by executing an assignment at Alice (both executions); next, it reduces by executing private decisions at Alice and Bob together to enter the then-branch (left execution) or else-branch (right); next, in the former case, it reduces by executing an assignment at Bob and terminates, whereas in the latter case, it terminates. Regarding concrete executions, two situations are possible:

– If B.y is initially 6, then the left abstract execution *can* induce a deadlock-free concrete one: after the first concrete reduction, A.x is 5, and B.y is still 6, so A.x==5 ∧ B.y==6 is true (i.e., case A, unanimity), *enabling* the sequel.
– If B.y is initially not 6, then both abstract executions *cannot* induce a deadlock-free concrete one: after the first concrete reduction, A.x is 5, but B.y is still not 6, so both A.x==5 ∧ B.y==6 and ¬A.x==5 ∧ ¬B.y==6 are false (i.e., case C, non-unanimity), *disabling* the sequel and causing a deadlock.

This example shows that we need a technique to infer that B.y must initially be 6 to ensure unanimity for deadlock freedom; we present it in the next subsection.

We end this subsection with an extension of ✓ for if/while-statements; it is induced by the rules in Fig. 6b (page 12). There is a third aim now (cf. page 12):

3. Rules [✓-If] and [✓-While] ensure that every role (in R) has its own conjunct in every multiparty condition. The idea is that every role always needs to know which branch to enter, so it must participate in every decision.[5,6]

[5] Well-formedness (every role has its own conjunct) and the grammar of if/while-statements (every conjunct is local to a role) are jointly similar to the *variable-knowledge-condition* of Neykova et al. [42]; they ensure that formulas are projectable (Fig. 10b).

[6] It is possible to encode choices in which only a few—not all—roles participate using extra variables; the idea is outlined at the end of Appx. A. However, this encoding

5.2 Predicate Transformer

We proceed with an extension of ϕ for if/while-statements; it is defined by the equations in Fig. 7b (page 13). As before, the definition of ϕ for if/while-statements is an adaptation of the definition of wp (i.e., Dijkstra's original predicate transformer [26]), but it differs on crucial points as well. More precisely:

- For **if** $\bigwedge\{e_r\}_{r \in R}\, G_1\, G_2$, the definition of ϕ has three conjuncts. The first (resp. second) conjunct states that if every e_r is true (resp. false), then the precondition of the then-branch (resp. else-branch) is true. This is similar to the definition of wp, and it includes case A (resp. B) on page 16.
  The third conjunct states that every e_{r_1} must imply every e_{r_2} (i.e., they are either all true or all false); this is new relative to the definition of wp, and it excludes case C on page 16. (i.e., if the precondition computed by ϕ is true, then case C will never arise). The following proposition makes this precise.

  Proposition 2. $[\![\bigwedge\{e_{r_1} \to e_{r_2}\}_{r_1, r_2 \in R}]\!] \subseteq [\![\bigwedge\{e_r\}_{r \in R} \vee \bigwedge\{\neg e_r\}_{r \in R}]\!]$.

  Thus, ϕ accumulates logical requirements not only to ensure the truth of the postcondition for functional correctness (i.e., the first and second conjunct), but also to ensure unanimity for deadlock freedom (i.e., the third conjunct). Figure 8c (page 14) shows an example, featuring the same global program as G on page 17: if $\phi_1[\mathsf{5/A.x}]$ is true (to ensure the truth of χ) and B.y is 6 (to ensure unanimity), then after executing the global program, χ is true. Thus, ϕ mechanises our reasoning about G on page 17.
- For **while** $\bigwedge\{e_r\}_{r \in R}\, \{\psi_{\text{inv}}\}\, G$, the definition of ϕ has an "outer conjunction" and an "inner conjunction". The inner conjunction is similar to ϕ for if-statements: either every e_r and the precondition of the body are true, to (re-)enter the loop, or every $\neg e_r$ and the postcondition are true, to exit it.
  The second outer conjunct states that *always* (i.e., in every state, i.e., before and after executing the body), if the invariant is true, then the inner conjunction is true; the first outer conjunct states that the invariant is indeed true (i.e., before executing the body). This is similar to the definition of wp.

5.3 Deadlock Freedom and Functional Correctness

To extend the main theorem for global programs (Thm. 3, page 16) to cover if/while-statements, we need to extend the auxiliary lemmas (Lem. 1–4, page 15 onwards); the proof of the theorem relies on the lemmas and is the same.

Lemma 5. *Lemmas 1–4 hold for this section's extension.*

Proof. For Lem. 1 there are no new cases (i.e., no new termination rules in Fig. 4b). For Lems. 2–3, the new cases (i.e., new reduction rules in Fig. 4b) can be proved directly. For Lem. 4, the new cases (i.e., new well-formedness rules in Fig. 6b) can be proved using Prop. 2, to establish that rule [→-IF1] or rule [→-IF2] is applicable in such a way that $\mathcal{S} \in [\![\psi]\!]$ holds as well. □

is not always practical/user-friendly. We therefore aim to offer "native" support for such choices too, using a form of *merging* [8,9,10]; see also Appx. D.

Theorem 4. *Theorem 3 holds for this section's extension.*

Proof. The same as the proof of Thm. 3, using Lem. 5 instead of Lems. 1–4. □

6 Global Programs: *Non-Blocking* If/While-Statements

In the previous section, we extended the base calculus of global programs with *blocking* if/while-statements; they require roles to make private decisions *together* (i.e., at the same time), using barriers. In this section, we extend the base calculus also with *non-blocking* if/while-statements; they allow roles to make private decisions *alone* (i.e., at their own pace). This is often preferable.

6.1 Syntax and Semantics

Recall that $\mathbb{G}$ denotes a universe of global programs, ranged over by G; it is induced by the following extended grammar:

$$G ::= \cdots \text{ (page 16)} \mid \textbf{if } \bigwedge\{e_r\}_{r\in R}\, G_1|_{R_1}\, G_2|_{R_2} \mid \textbf{while } \bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G|_{\emptyset}$$

Informally, the new grammar elements have the following meaning:[7]

- Global program **if** $\bigwedge\{e_r\}_{r\in R}\, G_1|_{R_1}\, G_2|_{R_2}$ prescribes a **non-blocking conditional choice** of G_1 and G_2. It relies on similar principles as the blocking version; notably, the same cases A, B, C on page 16 are applicable.
  The key difference with the blocking version is that roles make private decisions *alone* (i.e., at their own pace), without using synchronisation barriers. Intuitively, in operational terms, this means that for every role r: first, it privately evaluates its own conjunct e_r; next, it immediately enters a branch. To accommodate this, extra syntactic bookkeeping—in the form of the "$|_{R_1}$" and "$|_{R_2}$" notation—is needed to keep track of roles' decisions.
  More precisely, at any time, R_i contains every role that has already made a private decision to enter G_i. Initially, both R_1 and R_2 are empty. In case A (resp. B), R_1 (resp. R_2) eventually becomes "full" and contains all roles, while R_2 (resp. R_1) always remains empty. In case C, both R_1 and R_2 eventually become non-empty, but they always remain "non-full" as well.
  A non-blocking if-statement can terminate when all roles have made a private decision and every entered branch can terminate.
- Global program **while** $\bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G|_{\emptyset}$ prescribes a **non-blocking conditional loop** of G. The idea is similar to **if** $\bigwedge\{e_r\}_{r\in R}\, G_1|_{R_1}\, G_2|_{R_2}$, except that no extra bookkeeping is needed (i.e., a fixed $\emptyset$ in "$|_{\emptyset}$"): non-blocking while-statements will be unfolded into non-blocking if-statements. (The reason for the seemingly redundant "$|_{\emptyset}$" notation is to give non-blocking while-statements a different grammatical form than blocking ones.)

[7] Blocking and non-blocking if/while-statements have different syntax. This makes it possible to mix the blocking and non-blocking versions in the same global program (we have not encountered a compelling use case for this yet, though).

Formally, for non-blocking if/while-statements, $\downarrow$ and $\rightarrow$ are induced by the rules in Fig. 4c (page 10). Rule [$\downarrow$-NIF] states that if every role has made a private decision (left premise), and if G_1 and G_2 can terminate when at least one role has entered it (middle and right premise), then the non-blocking if-statement can terminate. The effect of the "$R_i \neq \emptyset$" conditions is that a non-entered branch does not need to be able to terminate for the whole if-statement to be able to terminate. We note that rule [$\downarrow$-NIF] also covers the case in which both R_1 and R_2 are non-empty, which should never have happened in the first place; shortly, we will rule it out using well-formedness and the predicate transformer.

Rules [$\rightarrow$-NIF1] and [$\rightarrow$-NIF2] state that if r has not made a private decision yet (left premise), then the non-blocking if-statement can reduce by executing one. For instance, when $G = \mathsf{A.x := 5}\,;\, \mathbf{if}\ (\mathsf{A.x{=}{=}5} \wedge \mathsf{B.y{=}{=}6})\ \mathsf{B.y := 7}|_{\emptyset}\ \mathbf{skip}|_{\emptyset}$ and $\psi = \mathsf{A.x{=}{=}5} \wedge \mathsf{B.y{=}{=}6}$, the following two abstract executions are possible:

$$G \xrightarrow{\substack{\mathtt{true},\\ \mathsf{A.x:=5}}} \bullet \xrightarrow{\substack{\mathsf{A.x{=}{=}5},\\ 1^{\{\mathsf{A}\}}}} \bullet \xrightarrow{\substack{\mathsf{B.y{=}{=}6},\\ 1^{\{\mathsf{B}\}}}} \bullet \xrightarrow{\substack{\mathtt{true},\\ \mathsf{B.y:=7}}} \mathbf{skip}\,;\, \mathbf{if}\ \psi\ \mathbf{skip}|_{\{\mathsf{A,B}\}}\ \mathbf{skip}|_{\emptyset}\downarrow$$

$$G \xrightarrow{\substack{\mathtt{true},\\ \mathsf{A.x:=5}}} \bullet \xrightarrow{\substack{\mathsf{A.x{=}{=}5},\\ 1^{\{\mathsf{A}\}}}} \bullet \xrightarrow{\substack{\neg\mathsf{B.y{=}{=}6},\\ 2^{\{\mathsf{B}\}}}} \mathbf{skip}\,;\, \mathbf{if}\ \psi\ \mathsf{B.y := 7}|_{\{\mathsf{A}\}}\ \mathbf{skip}|_{\{\mathsf{B}\}}$$

First, G reduces twice by executing an assignment and a private decision at Alice alone to enter the then-branch (both executions); next, it reduces by executing a private decision at Bob alone to enter the then-branch (top execution) or else-branch (bottom); next, in the latter case, it is stuck. Regarding concrete executions, if $\mathsf{B.y}$ is initially not $\mathsf{6}$, then a deadlock-free one does not exist: the top abstract execution *cannot* be enriched (i.e., after the second reduction, the sequel is disabled); the bottom abstract execution *can* be enriched but gets stuck. We note that unlike rules [$\rightarrow$-IF1] and [$\rightarrow$-IF2], there is no direct correspondence between rules [$\rightarrow$-NIF1] and [$\rightarrow$-NIF2] and cases A, B, C on page 16.

Rules [$\rightarrow$-NIF3] and [$\rightarrow$-NIF4] state that if G_1 or G_2 can reduce by executing γ (left premise), and if the subjects of γ have previously entered G_1 or G_2 (right premise), then the non-blocking if-statement can reduce accordingly. This means that global actions in the branches can be executed already before all private decisions have been made, out-of-order. We note that the set differences in the premises of these rules are needed, because in general (but undesirably), R_1 and R_2 may overlap; shortly, we will rule out this possibility using well-formedness and the predicate transformer. For instance, with the same G as above, also the following abstract execution is possible (due to rule [$\rightarrow$-SEQ2] as well):

$$G \xrightarrow{\substack{\mathtt{true},\\ \mathsf{A.x:=5}}} \bullet \xrightarrow{\substack{\mathsf{B.y{=}{=}6},\\ 1^{\{\mathsf{B}\}}}} \bullet \xrightarrow{\substack{\mathtt{true},\\ \mathsf{B.y:=7}}} \bullet \xrightarrow{\substack{\mathsf{A.x{=}{=}5},\\ 1^{\{\mathsf{A}\}}}} \mathbf{skip}\,;\, \mathbf{if}\ \psi\ \mathbf{skip}|_{\{\mathsf{A,B}\}}\ \mathbf{skip}|_{\emptyset}\downarrow$$

Rule [$\rightarrow$-NWHILE] unfolds the non-blocking while-statement.

We end this subsection with an extension of $\checkmark$ for non-blocking if/while-statements; it is induced by the rules in Fig. 6c (page 12). There is a fourth aim now (cf. page 12 and page 17):

4. Rule [$\checkmark$-NIF] ensures that case A or B on page 16 applies, but not case C: when roles make private decisions alone, they must still be unanimous.

6.2 Predicate Transformer

For non-blocking if/while-statements, ϕ is defined by the equations in Fig. 7c (page 13). It is based on the extension for the blocking variants in Fig. 7b:

- For **if** $\bigwedge\{e_r\}_{r\in R}\, G_1|_{R_1}\, G_2|_{R_2}$, the definition of ϕ has four cases.
  If R_1 and R_2 are both empty, then no role has made a private decision to enter a branch yet, so the precondition is the same as for blocking if-statements (i.e., either choice is still possible). This shows that blocking and non-blocking if-statements are functionally equivalent in the following sense: to ensure that the same postcondition is true in the end, the same precondition must be true in the beginning.
  If R_i and R_j are empty and non-empty, then the roles in R_j have privately decided to enter G_j. Thus, the precondition of G_j must be true. Moreover, to ensure that the remaining roles in $R \setminus R_j$ will privately make the same decision to enter G_j, their conjuncts must be all true (if j=1) or all false (if j=2) as well. In this way, cases A and B on page 16 are included.
  If R_1 and R_2 are both non-empty, then roles have privately decided to enter both G_1 and G_2, which should never have happened. Thus, the precondition is `false`. In this way, case C on page 16 is excluded.
- For **while** $\bigwedge\{e_r\}_{r\in R}\, \{\psi_{\text{inv}}\}\, G|_{\emptyset}$, no role has made a private decision to (re)enter the loop or exit it yet, so the precondition is the same as for blocking while-statements. When the first role privately decides, the non-blocking while-statement is unfolded into a non-blocking if-statement.

6.3 Main Theorem: Deadlock Freedom and Functional Correctness

To extend the main theorem for global programs (Thm. 3, page 16) to cover non-blocking if/while-statements, we need to extend the auxiliary lemmas (Lem. 1–4, page 15 onwards); the proof of the theorem relies on the lemmas and is the same.

Lemma 6. *Lemmas 1–4 hold for this section's extension.*

Proof. For Lem. 1, the new case (i.e., rule [↓-NIF] in Fig. 4c) can be proved using $\checkmark_R(G)$, to rule out the degenerate case that a non-blocking if-statement with the "empty" multiparty condition $\bigwedge\{e_r\}_{r\in\emptyset}$ can terminate. For Lem. 2, the new cases (i.e., new reduction rules in Fig. 4c) can be proved directly. For Lem. 3, the new cases (i.e., new reduction rules in Fig. 4c) can be proved using $\checkmark_R(G)$ and $[\![\phi(G,\chi)]\!] \neq \emptyset$ (first and second premise of Lem. 3), to establish that R_1 or R_2 is empty before the reduction, and remains empty after it (i.e., case C on page 16 never arises). For Lem. 4, the new cases (i.e., new well-formedness rules in Fig. 6b) can be proved using Prop. 2, to establish that rule [→-NIF1] or rule [→-NIF2] is applicable in such a way that $\mathcal{S} \in [\![\psi]\!]$ holds as well. □

Theorem 5. *Theorem 3 holds for this section's extension.*

Proof. The same as the proof of Thm. 3, using Lem. 6 instead of Lems. 1–4. □

7 Local Programs and Projection

In the previous sections, to cover the upper half of Fig. 1, we incrementally presented a calculus of global programs with blocking and non-blocking if/while-statements. In this section, to cover the bottom half, we present a complementary calculus of local programs and a projection function.

7.1 Syntax and Semantics

Let $\mathbf{\Lambda}$ and $\mathbb{L}$ denote universes of *local actions* and *local programs*, ranged over by λ and G; they are induced by the following grammar:

$$\begin{aligned} \lambda &::= q.y{:=}e \mid pq!e \mid pq?e \mid i_r^R \mid \tau \\ L &::= \mathbf{skip} \mid \lambda \mid L_1 \,;\, L_2 \mid L_1 \parallel L_2 \mid \\ &\quad\; R.\mathbf{if}\; e\; L_1\; L_2 \mid R.\mathbf{while}\; e\; L \mid \mathbf{if}\; e|_n\; L_1|_{R_1}\; L_2|_{R_2} \mid \mathbf{while}\; e|_n\; L|_{\emptyset} \end{aligned}$$

Informally, these grammar elements have the following meaning:

- Local action $q.y{:=}e$ models an **assignment**, as before.
- Local actions $pq!e$ and $pq?q$ model a **send** and a **receive** of the value of expression e at role p into variable y at role q.
- Local action i_r^R, with $i \in \{1, 2\}$, models a **private decision** at role r, as part of a collection of private decisions at all roles in R together.
- Local action τ models a **delay** (i.e., passage of time in which a role sits idle).
- The local programs have largely the same meaning as their global counterparts. There are two differences. First, the extra "$R.$" notation in blocking if/while-statements allows a role to know which other roles to wait for before entering a branch. Second, the extra "$|_n$" notation in non-blocking if/while-statements allows a role to delay n times (motivated below).

Formally, the abstract termination and reduction relations for local programs are induced by the same rules as in Fig. 4 (page 10), *mutatis mutandis*, except:

- In the rules for if/while-statements: every "$\bigwedge\{e_r\}_{r \in R}$" and "$\bigwedge\{\neg e_r\}_{r \in R}$" is replaced with "$e$" and "$\neg e$", while every "$i^R$" and "$i^{\{r\}}$" is replaced with "$i_r^R$" and "$i_r^{\{r\}}$" such that e is local to r. See Appx. B for details.
- There is an extra rule for non-blocking if-statements to execute a delay and decrement n if $n{>}0$ (motivated below, when discussing projection).

Let $\mathbb{R} \rightharpoonup \mathbb{L}$ denote a universe of *families of local programs* (i.e., partial functions roles to local programs), ranged over by $\mathcal{L}$. Informally, $\mathcal{L}$ prescribes a **parallel composition** of the k local programs in its image $\mathcal{L}(r_1), \ldots, \mathcal{L}(r_k)$. Formally, the abstract termination and reduction relations are induced by the rules in Fig. 9. They state that an assignment and a delay are executed alone, while a send–receive pair and a collection of private decisions are executed together. We note that for $n{=}1$, the bottom-left rule to execute $i^{\{r_1,\ldots,r_n\}}$ covers the case of non-blocking if/while-statements. Furthermore, the mechanisms by

$$\frac{\mathcal{L}(r_1)\downarrow \quad \cdots \quad \mathcal{L}(r_n)\downarrow}{\mathcal{L}\downarrow} \qquad \frac{\mathcal{L}(q) \xrightarrow{\psi, q.y:=e} L'_q}{\mathcal{L} \xrightarrow{\psi, q.y:=e} \mathcal{L}[q \mapsto L'_q]} \qquad \frac{\mathcal{L}(p) \xrightarrow{\psi_p, pq!e} L'_p \quad \mathcal{L}(q) \xrightarrow{\psi_q, pq?y} L'_q}{\mathcal{L} \xrightarrow{\psi_p \wedge \psi_q, p.e \twoheadrightarrow q.y} \mathcal{L}[p \mapsto L'_p, q \mapsto L'_q]}$$

$$\frac{\mathcal{L}(r_1) \xrightarrow{\xi_{r_1}, i^{\{r_1,\ldots,r_n\}}_{r_1}} L'_{r_1} \quad \cdots \quad \mathcal{L}(r_n) \xrightarrow{\xi_{r_n}, i^{\{r_1,\ldots,r_n\}}_{r_n}} L'_{r_n}}{\mathcal{L} \xrightarrow{\xi_{r_1} \wedge \cdots \wedge \xi_{r_n}, i^{\{r_1,\ldots,r_n\}}} \mathcal{L}[r_1 \mapsto L'_{r_1}, \ldots, r_n \mapsto L'_{r_n}]} \qquad \frac{\mathcal{L}(r) \xrightarrow{\psi,\tau} L'_r}{\mathcal{L} \xrightarrow{\psi,\tau} \mathcal{L}[r \mapsto L'_r]}$$

Fig. 9: Abstract operational semantics of families of local programs. $\mathcal{L}[r \mapsto L'_r]$ denotes the update of the image of r in $\mathcal{L}$ to L'_r.

$$q.y{:=}e \upharpoonright r = \begin{cases} q.y{:=}e & \text{if: } r = q \\ \tau & \text{otherwise} \end{cases} \qquad p.e \twoheadrightarrow q.y \upharpoonright r = \begin{cases} pq!e & \text{if: } r = p \\ pq?y & \text{if: } r = q \\ \tau & \text{otherwise} \end{cases} \qquad i^R \upharpoonright r = \begin{cases} i^R_r & \text{if: } r \in R \\ \tau & \text{otherwise} \end{cases}$$

(a) Global actions

$$\begin{aligned} \mathbf{skip} \upharpoonright r &= \mathbf{skip} \\ G_1 \circ G_2 \upharpoonright r &= (G_1 \upharpoonright r) \circ (G_2 \upharpoonright r) \\ \mathbf{if}\ \psi\ G_1\ G_2 \upharpoonright r &= R.\mathbf{if}\ (\psi \upharpoonright r)\ (G_1 \upharpoonright r)\ (G_2 \upharpoonright r) \\ \mathbf{while}\ \psi\ \{\psi_{\text{inv}}\}\ G \upharpoonright r &= R.\mathbf{while}\ (\psi \upharpoonright r)\ (G \upharpoonright r) \\ \mathbf{if}\ \psi\ G_1|_{R_1}\ G_2|_{R_2} \upharpoonright r &= \mathbf{if}\ (\psi \upharpoonright r)|_{|R \setminus (R_1 \cup R_2 \cup \{r\})|}\ (G_1 \upharpoonright r)|_{R_1 \cap \{r\}}\ (G_2 \upharpoonright r)|_{R_2 \cap \{r\}} \\ \mathbf{while}\ \psi\ \{\psi_{\text{inv}}\}\ G|_{\emptyset} \upharpoonright r &= \mathbf{while}\ (\psi \upharpoonright r)|_{|R \setminus \{r\}|}\ (G \upharpoonright r)|_{\emptyset} \end{aligned} \qquad \bigwedge\{e_{\hat{r}}\}_{\hat{r} \in R} \upharpoonright r = \begin{cases} e_r & \text{if: } r \in R \\ \mathtt{true} & \text{otherwise} \end{cases}$$

(b) Global programs. Let $\circ \in \{;, \|\}$ and $r \in R$.

Fig. 10: Decomposition of global actions/programs into local actions/programs

which "togetherness" arises (i.e., channels and barriers) are left implicit; they are implementation details. The concrete termination and reduction relations are induced by the same rules as in Fig. 5 (page 11), *mutatis mutandis*.

To decompose global actions and programs into local ones, let $\upharpoonright$ denote a *projection function*; it is induced by the equations in Fig. 10. In words, $\upharpoonright$ consumes a global program G and a role r as input, and it produces a local program $G \upharpoonright r$ as output. The idea is that $\upharpoonright$ is *sound* and *complete*: roughly, G can terminate or reduce by executing γ if, and only if, $G \upharpoonright r$ can similarly terminate or reduce by executing $\gamma \upharpoonright r$. The interesting cases of Fig. 10 are as follows:

- For γ (any global action), there are basically two possibilities. If r is a subject of γ, then $\gamma \upharpoonright r$ is the contribution of r to γ (i.e., an assignment remains an assignment; a communication is split into a separate send and receive; a collection of private decisions is split into separate ones). If r is not a subject of γ, then $\gamma \upharpoonright r$ is a delay (i.e., r sits idle, without contributing to γ).
- For $G = \mathbf{if}\ \psi\ G_1|_{R_1}\ G_2|_{R_2}$, the definition of $\upharpoonright$ is most complicated. We explain it from the perspective of soundness. There are three situations to consider.

First, suppose that G reduces by executing a global action γ in which r does participate. To ensure that $G \upharpoonright r$ can similarly reduce by executing $\gamma \upharpoonright r$, it will be sufficient to register in $G \upharpoonright r$ whether or not r has already entered a branch in G (and which one). This is achieved by "$|_{R_1 \cap \{r\}}$" and "$|_{R_2 \cap \{r\}}$". Second, suppose that G reduces by executing a global action $i^{\{r_0\}}$ in which r does not participate, using rule [→-NIF1] or rule [→-NIF2], so another role r_0 enters G_1 or G_2. To ensure that $G \upharpoonright r$ can similarly reduce by executing τ, it will be sufficient to register in $G \upharpoonright r$ the number of roles that have not yet entered a branch in G, excluding r. This is achieved by "$|_{|R \setminus (R_1 \cup R_2 \cup \{r\})|}$". Third, suppose that G reduces by executing a global action γ in which r does not participate using rule [→-NIF3] or rule [→-NIF4]. To ensure that $G \upharpoonright r$ can similarly reduce, no additional information needs to be registered.

7.2 Operational Equivalence

Informally, our main theorem for local programs and projection is as follows: if the global program is well-formed, and if the computed precondition is true in the initial state, then operational equivalence is provided. In the rest of this section, we first present auxiliary lemmas; next, we present the main theorem.

The first lemma pertains to soundness of $\upharpoonright$. It states that if G is well-formed and can terminate or reduce, then $G \upharpoonright r$ can similarly terminate or reduce.

Lemma 7.

1. *If* $\checkmark_R(G)$ *and* $r \in R$ *and* $G\downarrow$, *then* $(G \upharpoonright r)\downarrow$.
2. *If* $\checkmark_R(G)$ *and* $r \in R$ *and* $G \xrightarrow{\psi,\gamma} G'$, *then* $(G \upharpoonright r) \xrightarrow{\psi\upharpoonright r,\gamma\upharpoonright r} (G' \upharpoonright r)$.

Proof. By induction on the derivation of $G\downarrow$ (item 1) and $G \xrightarrow{\psi,\gamma} G'$ (item 2). The interesting cases are rules [→-IF1], [→-IF2], [→-WHILE1], and [→-WHILE2]: in those cases, we use premises $\checkmark_R(G)$ and $r \in R$ to establish that r must have its own conjunct in the multiparty condition, so it must contribute to γ. □

The second lemma pertains to completeness of $\upharpoonright$. It states that if G is well-formed, and if $G \upharpoonright r$ can terminate, then G can similarly terminate. Furthermore, it states that if G is well-formed, and if every $G \upharpoonright r$ can reduce by executing $\gamma \upharpoonright r$, for every subject r of γ, then G can similarly reduce.

Lemma 8.

1. *If* $\checkmark_R(G)$ *and* $(G \upharpoonright r)\downarrow$, *then* $G\downarrow$.
2. *If* $\checkmark_R(G)$ *and* $(G \upharpoonright r) \xrightarrow{\psi_r,\gamma\upharpoonright r} L'_r$, *for every* $r \in \mathsf{subj}(\gamma)$, *then* $G \xrightarrow{\psi,\gamma} G'$ *and* $\psi_r = \psi \upharpoonright r$ *and* $L'_r = G' \upharpoonright r$, *for every* r, *for some* ψ, G'.

Proof. By induction on the derivation of $(G \upharpoonright r)\downarrow$ (item 1) and the derivations of $(G \upharpoonright r) \xrightarrow{\psi_r,\gamma\upharpoonright r} L'_r$, for every $r \in \mathsf{subj}(\gamma)$ (item 2). The interesting cases are [→-PAR1] and [→-PAR2]: we use premise $\checkmark_R(G)$ to establish that either the LHS is reduced in every $G \upharpoonright r$, or the RHS (otherwise, there is no unique G'). □

Thus, the previous lemmas show that a global program and its family of projections can simulate each other's behaviour, at the abstract "top layer" of the operational semantics. The following theorem shows that this result can be extended to the concrete "bottom layer": it states that if G is well-formed, and if $\phi(G, \chi)$ is true in $\mathcal{S}$, then $(G, \mathcal{S})$ and $(\{G \upharpoonright r\}_{r \in R}, \mathcal{S})$ are *weakly bisimilar* (e.g., [30]), denoted with $\approx$. This means that $(G, \mathcal{S})$ and $(\{G \upharpoonright r\}_{r \in R}, \mathcal{S})$ can *coinductively* simulate each other's behaviour, modulo delays (i.e., **operational equivalence**).

Theorem 6. *If* $\checkmark_R(G)$ *and* $\mathcal{S} \in [\![\phi(G, \chi)]\!]$, *then* $(G, \mathcal{S}) \approx (\{G \upharpoonright r\}_{r \in R}, \mathcal{S})$.

Proof. We prove the theorem using Lems. 7–8 and Fig. 5. See Appx. C for a more detailed overview of the steps, including a *weak bisimulation relation.* □

8 Conclusion

We presented a new theory of choreographic programming. It supports for the first time: construction of distributed systems that require **decentralised decision making**; analysis of distributed systems to provide not only deadlock freedom but also **functional correctness**. Both contributions are enabled by a single new technique, namely *a predicate transformer for choreographies.*

The following corollary summarises our main theorems (Thms. 3–6):

Corollary 1. *If global program* G *(with* multiparty conditions *in if/while-statements) is well-formed, and if precondition* $\phi(G, \chi)$ *is true in initial state* $\mathcal{S}$, *then the family of projections* $(\{G \upharpoonright r\}_{r \in R}, \mathcal{S})$ *is deadlock-free and functionally-correct.*

For instance, in Sect. 2, we presented a deadlock-free global program for leader election; in Appx. E, we demonstrate how to prove its functional correctness; by Cor. 1, these properties are preserved by projection.

We implemented the new theory on top of the existing *VerCors* tool for deductive verification [4]; we present this implementation elsewhere.

In future work, we aim to extend the new theory with: **(1)** *asynchronous communication*; **(2)** a new version of *merging* [8,9,10] for decentralised decision making (see also footnote 6); **(3)** more *flexible interleaving* by relaxing the disjointness requirement for interleaving to support shared variables (e.g., using concurrent separation logic [6,43]).

Acknowledgments Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103.

References

1. Apt, K.R., Olderog, E.: Fifty years of hoare's logic. Formal Aspects Comput. **31**(6), 751–807 (2019)
2. Baeten, J.C.M., Bravetti, M.: A ground-complete axiomatisation of finite-state processes in a generic process algebra. Mathematical Structures in Computer Science **18**(6), 1057–1089 (2008)
3. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: POPL. pp. 191–202. ACM (2012)
4. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The vercors tool set: Verification of parallel and concurrent software. In: IFM. Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2017)
5. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 162–176. Springer (2010)
6. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1-3), 227–270 (2007)
7. Carbone, M., Cruz-Filipe, L., Montesi, F., Murawska, A.: Multiparty classical choreographies. In: LOPSTR. Lecture Notes in Computer Science, vol. 11408, pp. 59–76. Springer (2018)
8. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: ESOP. Lecture Notes in Computer Science, vol. 4421, pp. 2–17. Springer (2007)
9. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012)
10. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL. pp. 263–274. ACM (2013)
11. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. In: CONCUR. Lecture Notes in Computer Science, vol. 8704, pp. 47–62. Springer (2014)
12. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distributed Comput. **31**(1), 51–67 (2018)
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science **26**(2), 238–302 (2016)
14. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: FoSSaCS. Lecture Notes in Computer Science, vol. 10203, pp. 424–440 (2017)
15. Cruz-Filipe, L., Montesi, F.: Choreographies in practice. In: FORTE. Lecture Notes in Computer Science, vol. 9688, pp. 114–123. Springer (2016)
16. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. In: FACS. Lecture Notes in Computer Science, vol. 10231, pp. 17–35 (2016)
17. Cruz-Filipe, L., Montesi, F.: Encoding asynchrony in choreographies. In: SAC. pp. 1175–1177. ACM (2017)
18. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 92–107. Springer (2017)
19. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020)
20. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Communications in choreographies, revisited. In: SAC. pp. 1248–1255. ACM (2018)

21. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: ICTAC. Lecture Notes in Computer Science, vol. 12819, pp. 115–133. Springer (2021)
22. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: ITP. LIPIcs, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
23. Deniélou, P., Yoshida, N.: Dynamic multirole session types. In: POPL. pp. 435–446. ACM (2011)
24. Deniélou, P., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: ICALP (2). Lecture Notes in Computer Science, vol. 7966, pp. 174–186. Springer (2013)
25. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Logical Methods in Computer Science **8**(4) (2012)
26. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
27. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor. Comput. Sci. **328**(1-2), 19–37 (2004)
28. Giallorenzo, S., Montesi, F., Gabbrielli, M.: Applied choreographies. In: FORTE. Lecture Notes in Computer Science, vol. 10854, pp. 21–40. Springer (2018)
29. Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty languages: The choreographic and multitier cases (pearl). In: ECOOP. LIPIcs, vol. 194, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
30. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)
31. Hildebrandt, T.T., Slaats, T., López, H.A., Debois, S., Carbone, M.: Declarative choreographies and liveness. In: FORTE. Lecture Notes in Computer Science, vol. 11535, pp. 129–147. Springer (2019)
32. Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris: session-type based reasoning in separation logic. Proc. ACM Program. Lang. **4**(POPL), 6:1–6:30 (2020)
33. Hoare, C.A.R.: Parallel programming: An axiomatic approach. Comput. Lang. **1**(2), 151–160 (1976)
34. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998)
35. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
36. Hurlin, C.: Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic. (Spécification et vérification de programmes orientés objets en logique de séparation). Ph.D. thesis, University of Nice Sophia Antipolis, France (2009)
37. Itai, A., Rodeh, M.: Symmetry breaking in distributive networks. In: FOCS. pp. 150–158. IEEE Computer Society (1981)
38. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. Comput. **88**(1), 60–87 (1990)
39. Jongmans, S.S., van den Bos, P.: A Predicate Transformer for Choreographies (Technical Report). Tech. Rep. OUNL-CS-2022-02, Open University of the Netherlands (2022)
40. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: OOPSLA. pp. 280–298. ACM (2015)

41. Montesi, F., Yoshida, N.: Compositional choreographies. In: CONCUR. Lecture Notes in Computer Science, vol. 8052, pp. 425–439. Springer (2013)
42. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: CC. pp. 128–138. ACM (2018)
43. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1-3), 271–307 (2007)
44. Peleg, D.: Time-optimal leader election in general networks. J. Parallel Distributed Comput. **8**(1), 96–99 (1990)
45. Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies - safe runtime updates of distributed applications. In: COORDINATION. Lecture Notes in Computer Science, vol. 9037, pp. 67–82. Springer (2015)
46. Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies: Theory and implementation. Log. Methods Comput. Sci. **13**(2) (2017)
47. Preda, M.D., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M.: AIOCJ: A choreographic framework for safe adaptive distributed applications. In: SLE. Lecture Notes in Computer Science, vol. 8706, pp. 161–170. Springer (2014)
48. Rensink, A., Wehrheim, H.: Process algebra with action dependencies. Acta Informatica **38**(3), 155–234 (2001)
49. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
50. Skeen, D.: Nonblocking commit protocols. In: SIGMOD Conference. pp. 133–142. ACM Press (1981)
51. Toninho, B., Yoshida, N.: Certifying data in multiparty session types. J. Log. Algebraic Methods Program. **90**, 61–83 (2017)
52. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 148:1–148:30 (2020)

```
1. C.files := emptyList() ;
2. C.names := [v1,...,vn] ;
3. C.names →▷ S.names ;
4. S.i := 0 ;
5. while S.(i<size(names))
6.   (S.CONTINUE →▷ C ;
7.    S.file(get(names,i)) →▷ C.f ;
8.    C.files := append(files,f) ;
9.    S.i := i+1) ;
10. S.END →▷ C
```

(a) Centralised

```
1. C.files := emptyList() ;
2. C.names := [v1,...,vn] ;
3. C.names →▷ S.names ;
4. S.i := 0 ; C.i := 0 ;
5. while S.(i<size(names))
6.       ∧ C.(i<size(names))
7.   (S.file(get(names,i)) →▷ C.f ;
8.    C.files := append(files,f)
9.    S.i := i+1 ; C.i := i+1)
```

(b) Decentralised

Fig. 11: Global programs for batch file transfer (Exmp. 3)

A Additional Motivating Examples

Regarding the usefulness of the new theory, the following example shows that centralised decision making can be *impractical* (i.e., unnatural or unnecessary).

Example 3 (Batch file transfer). Consider a distributed system in which two processes enact roles Client and Server.

First, a list of n file names is communicated from Client to Server. Next, every file in the list is communicated from Server to Client. This kind of functionality is supported by many popular file transfer tools, including `curl`, `wget`, and `ftp`.

Figure 11 shows two global programs: one that uses centralised decision making (at Server), and one that uses the new theory's decentralised decision making.

With centralised decision making, first, Client initialises its variables `files` and `names` (lines 1–2). Next, `names` is communicated from Client to Server (line 3). We note that Client and Server each have a variable `names`, but while the identifiers are the same, the variables are different and inhabit disjoint memory spaces. Next, Server initialises loop variable `i` (line 4), decides if another file needs to be transferred (line 5), and informs Client accordingly (lines 6 and 10). If indeed needed, a file is communicated from Server to Client (line 7), Client appends the file to `files`, and Server increments `i` (line 9).[8]

The issue with centralised decision making in this example is that it is *unnatural* (arguably) and *unnecessary* (definitely) for Server to explicitly inform Client about how to proceed using label communications: already before starting the loop, *both Client and Server* know how many iterations will ensue, namely n. Thus, by letting *both Client and Server* count the iterations (each using its own `i`), decentralised decision making can be leveraged to avoid the unnecessary label communications. This is shown in Fig. 11b; the additions are highlighted. □

Regarding the necessity of the new theory, the following example shows that centralised decision making can be *impossible*.

[8] Many existing calculi of global programs do not have explicit while-statements, but if-statements and recursion, which can be used to encode while-statements.

```
1. (S1.db:=newTx(db,query1) ||
2.  S2.db:=newTx(db,query2)) ;
3. (S1.status(db) ↠ C.x1 ||
4.  S2.status(db) ↠ C.x2) ;
5. if C.x1==x2==true
6.    (C.SUCC ↠ S1 ; S1.db:=commit(db) ||
7.     C.SUCC ↠ S2 ; S2.db:=commit(db))
8.    (C.FAIL ↠ S1 ; S1.db:=abort(db) ||
9.     C.FAIL ↠ S2 ; S2.db:=abort(db))
```

(a) Centralised

```
1. (S1.db:=newTx(db,query1) ||
2.  S2.db:=newTx(db,query2)) ;
3. (S1.status(db) ↠ S2.x ||
4.  S2.status(db) ↠ S1.x) ;
5. if ⋀{r.status(db)==x==true}r∈{S1,S2}
6.    (S1.db:=commit(db) ||
7.     S2.db:=commit(db))
8.    (S1.db:=abort(db) ||
9.     S2.db:=abort(db))
```

(b) Decentralised

Fig. 12: Global programs for two-phase commit in distributed databases (Exmp. 4)

Example 4 (Coordinator-less two-phase commit in distributed databases). Consider a distributed system in which two processes enact roles Server1 and Server2, each of which holds a share of a distributed database.

When two queries that constitute one *transaction* are processed by different Servers, special measures are needed to ensure *atomicity* (i.e., either both queries take effect, or none). A classical approach is to use a *two-phase commit* protocol: in the first phase, the Servers locally execute the queries and share a success/failure status flag; in the second phase, if both Servers succeeded, then they *commit*, or else they *abort*. Basically, there are two variants of this protocol, which differ in *with whom* the Servers share the status flags [50]: in the *coordinator-based* variant, an extra process enacts role Coordinator to receive the status flags, decide whether the Servers should commit or abort, and share the outcome; in the *coordinator-less* variant, without an extra process, the Servers receive each other's status flag and privately decide to commit or abort.

Figure 12 shows two global programs: one that uses centralised decision making, and one that uses the new theory's decentralised decision making. The former implements the coordinator-based two-phase commit protocol, while the latter implements the coordinator-less variant. We note: (1) in both global programs, the two shares of the distributed database are represented with variable `db` at Server1 and variable `db` at Server2; (2) the data structures in these variables also keep track of local transactions, which are started with `newTx`, inspected with `status`, and finished with `commit` and `abort`; (3) as in functional programming, functions are side-effect-free for simplicity, so `newTx`, `status`, `commit`, and `abort` return new data structures that reflect these functions' effects.

The point of this example is that the coordinator-less variant, in which the Servers are treated symmetrically [50], *cannot* faithfully be implemented using centralised decision making. The reason is that centralised decision making inherently requires the Servers to be treated asymmetrically. □

Regarding the generality of the new theory, we now illustrate that centralised decision making can be *encoded* as decentralised, by translating one-party con-

if $p.e$	$p.x := e$;
$(p.\texttt{THEN} \twoheadrightarrow q_1$; $\cdots$; $p.\texttt{THEN} \twoheadrightarrow q_k$; $G_{\text{then}})$	$p.x \twoheadrightarrow q_1.x$; $\cdots$; $p.x \twoheadrightarrow q_k.x$;
$(p.\texttt{ELSE} \twoheadrightarrow q_1$; $\cdots$; $p.\texttt{ELSE} \twoheadrightarrow q_k$; $G_{\text{else}})$	**if** $(p.x \wedge q_1.x \wedge \cdots \wedge q_k.x)$ G_{then} G_{else}
(a) One-party conditional choice	$\Longrightarrow$ (b) Multiparty conditional choice

Fig. 13: From one-party conditional choice to multiparty conditional choice

ditional choices (with label communications) into multiparty ones (without); a similar translation can be defined for conditional loops.

Figure 13 shows the translation. In the centralised case, role p evaluates condition e, enters a branch, and shares the outcome with roles $q_1, \ldots, q_k$ by communicating labels `THEN` or `ELSE`; thus, $q_1, \ldots, q_k$ enter the same branch. In the decentralised case, p still evaluates e, but instead of sending a label, it sends the "raw" boolean value; next, all roles privately check the value and enter the same branch.

The translation in the opposite direction does not work in general, because it requires the availability of a distinguished role p. As demonstrated in Exmp. 2 and Exmp. 4, when processes are symmetric, this requirement fails.

B Operational Semantics of Local Programs

See Fig. 14.

C Additional Proof Steps of Thm. 6

We start by introducing some notation. We write $(G, \mathcal{S}) \Downarrow$ and $(\mathcal{L}, \mathcal{S}) \Downarrow$ to indicate that these configurations can terminate, *after* 0-or-more delay reductions. Similarly, we write $(G, \mathcal{S}) \overset{\gamma^{\text{c}}}{\Longrightarrow} (G', \mathcal{S}')$ and $(\mathcal{L}, \mathcal{S}) \overset{\gamma^{\text{c}}}{\Longrightarrow} (\mathcal{L}', \mathcal{S}')$ to indicate that these configurations can reduce by executing γ^{c}, *after and before* 0-or-more delay reductions. Furthermore, we write $L_1 \Rrightarrow\!\!\Lleftarrow L_2$ to indicate that L_1 and L_2 can reach the same $L^\dagger$ after 0-or-more unconditional delay reductions. Using this last piece of notation, we introduce an auxiliary *correspondence relation* between configurations of global programs and families of local program (not necessarily projections), denoted by $\bowtie$; it is induced by the following rule:

$$\frac{\checkmark_{\text{dom}\,\mathcal{L}}(G) \quad \mathcal{S} \in [\![\phi(G, \chi)]\!] \quad (G \upharpoonright r) \Rrightarrow\!\!\Lleftarrow \mathcal{L}(r) \text{ for every } r \in \text{dom}\,\mathcal{L}}{(G, \mathcal{S}) \bowtie (\mathcal{L}, \mathcal{S})}\ [\bowtie]$$

This correspondence relation, as well as the other pieces of notation, allow us to prove the following lemma. It states that if $(G, \mathcal{S})$ and $(\mathcal{L}, \mathcal{S})$ correspond, then they can *coinductively* simulate each other's behaviour (modulo delays). That is: $\bowtie$ is a *weak bisimulation*, while $(G, \mathcal{S})$ and $(\mathcal{L}, \mathcal{S})$ are *weakly bisimilar* (e.g., [30]).

Lemma 9. *If* $(G, \mathcal{S}) \bowtie (\mathcal{L}, \mathcal{S})$*:*

$$\frac{}{\mathbf{skip}\downarrow} \qquad \frac{L_1\downarrow \quad L_2\downarrow}{L_1\,;\,L_2\downarrow} \qquad \frac{L_1\downarrow \quad L_2\downarrow}{L_1 \parallel L_2\downarrow} \qquad \frac{\psi = \mathtt{true}}{\lambda \xrightarrow{\psi,\lambda} \mathbf{skip}}$$

$$\frac{L_1 \xrightarrow{\psi,\lambda} L_1'}{L_1\,;\,L_2 \xrightarrow{\psi,\lambda} L_1'\,;\,L_2} \qquad \frac{\mathsf{subj}(L_1) \cap \mathsf{subj}(\lambda) = \emptyset \quad L_2 \xrightarrow{\psi,\lambda} L_2'}{L_1\,;\,L_2 \xrightarrow{\psi,\lambda} L_1\,;\,L_2'}$$

$$\frac{L_1 \xrightarrow{\psi,\lambda} L_1'}{L_1 \parallel L_2 \xrightarrow{\psi,\lambda} L_1' \parallel L_2} \qquad \frac{L_2 \xrightarrow{\psi,\lambda} L_2'}{L_1 \parallel L_2 \xrightarrow{\psi,\lambda} L_1 \parallel L_2'}$$

(a) Base calculus

$$\frac{\psi = e \quad \lambda = 1_r^R \quad \mathsf{subj}(e) = \{r\}}{R.\mathbf{if}\ e\ L_1\ L_2 \xrightarrow{\psi,\lambda} L_1} \qquad \frac{\psi = \neg e \quad \lambda = 2_r^R \quad \mathsf{subj}(e) = \{r\}}{R.\mathbf{if}\ e\ L_1\ L_2 \xrightarrow{\psi,\lambda} L_2}$$

$$\frac{\psi = e \quad \lambda = 1_r^R \quad \mathsf{subj}(e) = \{r\}}{R.\mathbf{while}\ e\ G \xrightarrow{\psi,\lambda} G\,;\,\mathbf{while}\ e\ G} \qquad \frac{\psi = \neg e \quad \lambda = 2_r^R \quad \mathsf{subj}(e) = \{r\}}{R.\mathbf{while}\ e\ G \xrightarrow{\psi,\lambda} \mathbf{skip}}$$

(b) Extension with if/while-statements

$$\frac{R = R_1 \cup R_2 \quad R_1 \neq \emptyset \text{ implies } L_1\downarrow \quad R_2 \neq \emptyset \text{ implies } L_2\downarrow}{\mathbf{if}\ e|_0\ L_1|_{R_1}\ L_2|_{R_2}\downarrow}$$

$$\frac{n > 0}{\mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2|_{R_2} \xrightarrow{\mathtt{true},\tau} \mathbf{if}\ e|_{n-1}\ L_1|_{R_1}\ L_2|_{R_2}}$$

$$\frac{r \in R \setminus (R_1 \cup R_2) \quad \psi = e \quad \lambda = 1_r^{\{r\}} \quad \mathsf{subj}(e) = \{r\}}{\mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2|_{R_2} \xrightarrow{\psi,\lambda} \mathbf{if}\ e|_n\ L_1|_{R_1 \cup \{r\}}\ L_2|_{R_2}}$$

$$\frac{r \in R \setminus (R_1 \cup R_2) \quad \psi = \neg e \quad \lambda = 2_r^{\{r\}} \quad \mathsf{subj}(e) = \{r\}}{\mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2|_{R_2} \xrightarrow{\psi,\lambda} \mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2|_{R_2 \cup \{r\}}}$$

$$\frac{L_1 \xrightarrow{\psi,\lambda} L_1' \quad \mathsf{subj}(\lambda) \subseteq R_1 \setminus R_2}{\mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2|_{R_2} \xrightarrow{\psi,\lambda} \mathbf{if}\ e|_n\ L_1'|_{R_1}\ L_2|_{R_2}}$$

$$\frac{L_2 \xrightarrow{\psi,\lambda} L_2' \quad \mathsf{subj}(\lambda) \subseteq R_2 \setminus R_1}{\mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2|_{R_2} \xrightarrow{\psi,\lambda} \mathbf{if}\ e|_n\ L_1|_{R_1}\ L_2'|_{R_2}}$$

$$\frac{\mathbf{if}\ e|_n\ (G\,;\,\mathbf{while}\ e|_n\ G|_\emptyset)|_\emptyset\ \mathbf{skip}|_\emptyset \xrightarrow{\psi,\lambda} G'}{\mathbf{while}\ e|_n\ G|_\emptyset \xrightarrow{\psi,\lambda} G'}$$

(c) Extension with non-blocking if/while-statements

Fig. 14: Abstract operational semantics of local programs ("top layer")

1. *If* $(G, \mathcal{S}) \Downarrow$, *then* $(\mathcal{L}, \mathcal{S}) \Downarrow$.
2. *If* $(G, \mathcal{S}) \xrightarrow{\gamma}_c (G', \mathcal{S}')$, *then* $(\mathcal{L}, \mathcal{S}) \xRightarrow{\gamma}_c (\mathcal{L}', \mathcal{S}')$ *and* $(G', \mathcal{S}') \bowtie (\mathcal{L}', \mathcal{S}')$, *for some* $\mathcal{L}'$.
3. *If* $(\mathcal{L}, \mathcal{S}) \Downarrow$, *then* $(G, \mathcal{S}) \Downarrow$.
4. *If* $(\mathcal{L}, \mathcal{S}) \xrightarrow{\gamma}_c (\mathcal{L}', \mathcal{S}')$, *then* $(G, \mathcal{S}) \xRightarrow{\gamma}_c (G', \mathcal{S}')$ *and* $(G', \mathcal{S}') \bowtie (\mathcal{L}', \mathcal{S}')$, *for some* G'.

Proof. Mimicry of termination and reduction in items 1–4 can be proved using $\checkmark_{\mathrm{dom}\,\mathcal{L}}(G)$ (first premise of rule [⋈]) and $(G \upharpoonright r) \Rightarrow\!\!\!\!\Leftarrow \mathcal{L}(r)$ for every $r \in \mathrm{dom}\,\mathcal{L}$ (third premise), by applying Lems. 7–8. Correspondence between G' and $\mathcal{L}'$ in items 2 and 4 can be proved in two parts: (re-)establishment of $\checkmark_{\mathrm{dom}\,\mathcal{L}}(G')$ and $\mathcal{S} \in [\![\phi(G', \chi)]\!]$ (first and second premise) can be proved using $\checkmark_{\mathrm{dom}\,\mathcal{L}}(G)$ and $\mathcal{S} \in [\![\phi(G, \chi)]\!]$ (first and second premise of rule [⋈]), by applying Prop. 1 and Lems. 2–3, plus their extensions in Lems. 5–6; (re-)establishment of $(G' \upharpoonright r) \Rightarrow\!\!\!\!\Leftarrow \mathcal{L}(r)$, for every $r \in \mathrm{dom}\,\mathcal{L}'$, can be proved by induction. □

The following lemma states that if G is well-formed, and if $\phi(G, \chi)$ is true in $\mathcal{S}$, then $(G, \mathcal{S})$ and $(\{G \upharpoonright r\}_{r \in R}, \mathcal{S})$ correspond.

Lemma 10. *If* $\checkmark_R(G)$ *and* $\mathcal{S} \in [\![\phi(G, \chi)]\!]$, *then* $(G, \mathcal{S}) \bowtie (\{G \upharpoonright r\}_{r \in R}, \mathcal{S})$.

Proof. Rule [⋈] can directly be applied. □

Theorem 6 immediately follows from Lems. 9–10.

D Why and How to Merge?

Let G denote a global program, and let R denote the set of roles that occur in G. For G to be well-formed relative to R (i.e., $\checkmark_R(G)$), rules [✓-If] and [✓-While] demand that every role in R has its own conjunct in every multiparty condition in G (i.e., every role always needs to know which branch to enter, so it must participate in every decision).

It is possible to encode choices in which only a few—not all—roles participate using extra variables; the idea is outlined at the end of Appx. A. For instance, consider the following ill-formed (because not all roles participate) global type:

$$
\begin{aligned}
G = {} & \mathsf{A.x} := 5 \,;\, \mathsf{B.y} := 6 \,; \\
& \textbf{if } (\mathsf{A.x{=}{=}5} \wedge \mathsf{B.y{=}{=}6}) \\
& \quad \mathsf{A.true} \twoheadrightarrow \mathsf{C1.z} \,;\, \mathsf{B.true} \twoheadrightarrow \mathsf{C2.z} \,;\, G_{\text{then}} \\
& \quad \mathsf{A.false} \twoheadrightarrow \mathsf{C1.z} \,;\, \mathsf{B.false} \twoheadrightarrow \mathsf{C2.z} \,;\, G_{\text{else}}
\end{aligned}
$$

In this example, only A and B participate in the decision; subsequently, A informs C1 of the outcome, and B informs C2. This ill-formed global type can be rewritten into the following "morally equivalent" well-formed version:

$$
\begin{aligned}
\hat{G} = {} & \mathsf{A.x} := 5 \,;\, \mathsf{B.y} := 6 \,; \\
& \mathsf{A.outcome} := \mathtt{x==5} \,;\, \mathsf{B.outcome} := \mathtt{y==6} \,; \\
& \mathsf{A.outcome} \rightarrowtriangle \mathsf{C1.outcome} \,;\, \mathsf{B.outcome} \rightarrowtriangle \mathsf{C2.outcome} \,; \\
& \textbf{if } \mathsf{A.outcome} \wedge \mathsf{B.outcome} \wedge \mathsf{C.outcome} \wedge \mathsf{C2.outcome} \\
& \quad G_{\text{then}} \\
& \quad G_{\text{else}}
\end{aligned}
$$

In this example, first, A and B evaluate their local conditions (before the if-statement) and share the outcomes with C1 and C2; next, everybody participates in the decision. The encoding works similarly for non-blocking if-statements. Notably, a non-blocking if-statement allows C1 to proceed with G_{then} before C2 has received the outcome, just as in the original G.

However, this encoding is not always practical/user-friendly. We therefore aim to offer "native" support for such choices too, using a form of *merging* [8,9,10]. There are two main challenges.

1. The first challenge is technical: to prove operational equivalence (modulo weak bisimilarity), our current proof method depends on a tight correspondence between reductions of a global program and reductions of its individual projections (Lem. 7 and Lem. 8); it crucially relies on the use of τ-actions. This tight correspondence breaks in the presence of merging, as it is unclear how to merge sequences of τ-actions in a satisfactory way. To alleviate this, we are exploring a theory in which sequences of τ-actions can be dynamically "backtracked", as an alternative to statically merging them, but this is still work-in-progress.
2. The second challenge is more "philosophical"; it pertains to while-statements. Consider the following example:

$$
\begin{aligned}
& \mathsf{A.x} := 5 \,;\, \mathsf{B.y} := 5 \,; \\
& \textbf{while } (\mathsf{A.x==5} \wedge \mathsf{B.y==5}) \, \{\psi_{\text{inv}}\} \, \mathsf{A.x} \rightarrowtriangle \mathsf{B.y} \\
& \mathsf{C.z} := \mathtt{true}
\end{aligned}
$$

   The question is whether or not the local assignment at C should be allowed to happen. Naively, as C does not occur in the first two lines, one can argue that it *should* be allowed to happen, at any time. In contrast, since the loop never terminates, one can alternatively argue that the third line should be interpreted as unreachable/dead code; in that case, it *should not* be allowed to happen. A third option is to consider this program ill-formed, but then well-formedness comes to rely on termination detection for loops. It is an open question which of these three options is preferable.

E Reasoning About Functional Correctness

To clarify how to reason about functional correctness using this paper's new theory, we revisit Exmp. 2 and demonstrate how to prove the postcondition that "exactly one leader is elected". The following formula encodes this property:

$$\begin{aligned}\chi = \quad & (\ \texttt{P1.leader} \wedge \neg\texttt{P2.leader} \wedge \neg\texttt{P3.leader}) \\ \vee\ & (\neg\texttt{P1.leader} \wedge\ \ \texttt{P2.leader} \wedge \neg\texttt{P3.leader}) \\ \vee\ & (\neg\texttt{P1.leader} \wedge \neg\texttt{P2.leader} \wedge\ \ \texttt{P3.leader})\end{aligned}$$

Or more compactly, using a standard shorthand $\oplus$ for exclusive-or:

$$\chi = \texttt{P1.leader} \oplus \texttt{P2.leader} \oplus \texttt{P3.leader}$$

To check that global program G in Fig. 3 brings about this postcondition, the following standard method of deductive verification can be applied:

1. Annotate all while-statements with a sufficiently strong loop invariant ψ_{inv}. Generally, as usual in deductive verification, this may require some creativity/ingenuity from the programmer. In this example, however, it is reasonably simple: we need to assert that before/after every iteration, all processes have consistent knowledge about all identifiers, and no process has marked itself as leader yet. Formally:

$$\begin{aligned}\psi_{\text{inv}} = \quad & \texttt{P1.id1==P2.id1==P3.id1} \\ \wedge\ & \texttt{P1.id2==P2.id2==P3.id2} \\ \wedge\ & \texttt{P1.id3==P2.id3==P3.id3} \\ \wedge\ & \neg\texttt{P1.leader} \wedge \neg\texttt{P2.leader} \wedge \neg\texttt{P3.leader}\end{aligned}$$

2. Compute precondition $\phi(G, \chi)$, where G is the entire global program; this is just a mechanical process that can be carried out completely automatically (given the loop invariants of step 1). For every initial state that satisfies $\phi(G, \chi)$, functional correctness (i.e., postcondition χ is brought about) is guaranteed. We note that in this example, we also need to separately prove the intended relation between `maxIsUnique` and `max` (formalised below). In practice, this is typically done using pre/postcondition contracts for functions. As contract-based reasoning about sequential code is already well-studied and orthogonal to the concurrency aspects that are central to this paper, we omitted it. That said, formally, the intended relation is this:

$$\begin{aligned}\texttt{maxIsUnique(x,y,z)} \rightarrow \quad & \texttt{max(x,y,z)==x} \\ \oplus\ & \texttt{max(x,y,z)==y} \\ \oplus\ & \texttt{max(x,y,z)==z}\end{aligned}$$

We note that these steps are not specific to choreographies or this paper's new theory; they are standard practice in deductive verification.