Dynamic Choreographies⋆
Safe Runtime Updates of Distributed Applications

Mila Dalla Preda[1], Maurizio Gabbrielli[2], Saverio Giallorenzo[2],
Ivan Lanese[2], and Jacopo Mauro[2]

[1] Department of Computer Science - Univ. of Verona
[2] Department of Computer Science and Engineering - Univ. of Bologna / INRIA

Abstract. Programming distributed applications free from communication deadlocks and races is complex. Preserving these properties when applications are updated at runtime is even harder.

We present DIOC, a language for programming distributed applications that are free from deadlocks and races by construction. A DIOC program describes a whole distributed application as a unique entity (choreography). DIOC allows the programmer to specify which parts of the application can be updated. At runtime, these parts may be replaced by new DIOC fragments from outside the application. DIOC programs are compiled, generating code for each site, in a lower-level language called DPOC. We formalise both DIOC and DPOC semantics as labelled transition systems and prove the correctness of the compilation as a trace equivalence result. As corollaries, DPOC applications are free from communication deadlocks and races, even in presence of runtime updates.

1 Introduction

Programming distributed applications is an error-prone activity. Participants send and receive messages and, if the application is badly programmed, participants may get stuck waiting for messages that never arrive (communication deadlock), or they may receive messages in an unexpected order, depending on the speed of the other participants and of the network (races).

Recently, language-based approaches have been proposed to tackle the complexity of programming concurrent and distributed applications. Languages such as Rust [21] or SCOOP [19] provide higher-level primitives to program concurrent applications which avoid by construction some of the risks of concurrent programming. Indeed, in these settings most of the work needed to ensure a correct behaviour is done by the language compiler and runtime support. Using these languages requires a conceptual shift from traditional ones, but reduces times and costs of development, testing, and maintenance by avoiding some of the most common programming errors.

Here, we propose an approach based on *choreographic programming* [4, 5, 15, 22] following a similar philosophy, tailored for distributed applications. In choreographic programming, a whole distributed application is described as a unique entity, by specifying the expected interactions and their order. For instance, a price request from a buyer to a seller is written as `priceReq: buyer( b_prod )` → `seller( s_prod )`. It specifies that the `buyer` sends along channel `priceReq` the name of the desired product `b_prod` to the `seller`, which stores it in its local variable `s_prod`. Since in choreographic languages sends and receives are always paired, the coupling of exactly one receive with each send and vice versa makes communication deadlocks or races impossible to write. Given a choreography, a main challenge is to produce low-level distributed code which correctly implements the desired behaviour.

We take this challenge one step forward: we consider *updatable* applications, whose code can change while the application is running, dynamically integrating code from the outside. Such a feature, tricky in a sequential setting and even more in a distributed one, has countless uses: deal with emergency requirements, cope with rules and requirements which depend on contextual properties, improve and specialize the application to user preferences, and so on. We propose a general mechanism, which consists in delimiting inside the application blocks of code, called *scopes*, that may be dynamically replaced with new code, called *update*. The details of the behaviour of the updates do not need to be foreseen, updates may even be written while the application is running.

Runtime code replacement performed using languages not providing dedicated support is extremely error-prone. For instance, considering the price request example above, assume that we want to update the system allowing the buyer to send to the seller also its fidelity card ID to get access to some special offer. If the buyer is updated first and it starts the interaction before the seller has been updated, the seller is not expecting the card ID, which may be sent and lost, or received later on, when some different message is expected, thus breaking the correctness of the application. Vice versa, if the seller is updated first, (s)he will wait for the card ID, which the buyer will not send, leading the application to a deadlock. In our setting, the available updates may change at any time, posing an additional challenge. Extra precautions are needed to ensure that all the participants agree on which code is used for a given update. For instance, in the example above, suppose that the buyer finds the update that allows the sending of the card ID, and applies this update before the seller does. If the update is no more available when the seller looks for it, then the application ends up in an inconsistent state, where the update is only partially applied, and the seller will receive an unexpected message containing the card ID.

If both the original application and the updates are programmed using a choreographic language, these problems cannot arise. In fact, at the choreographic level, the update is applied atomically to all the involved participants. Again, the tricky part is to compile the choreographic code to low-level distributed code ensuring correct behaviour. In particular, at low-level, the differ-

ent participants have to coordinate their updates avoiding inconsistencies. The present paper proposes a solution to this problem. In particular:

- we define a choreographic language, called DIOC, to program distributed applications and supporting code update (§ 2);
- we define a low-level language, called DPOC, based on standard send and receive primitives (§ 3);
- we define a behaviour-preserving projection function compiling DIOCs into DPOCs (§ 3.1);
- we give a formal proof of the correctness of the projection function (§ 4). Correctness is guaranteed even in a scenario where the new code used for updates dynamically changes at any moment and without notice.

The contribution outlined above is essentially theoretical, but it has already been applied in practice, resulting in AIOCJ, an adaptation framework described in [8]. The theoretical underpinning of AIOCJ is a specific instantiation of the results presented here. Indeed, AIOCJ further specifies how to manage the updates, e.g., how to decide when updates should be applied and which ones to choose if many of them apply. For more details on the implementation and more examples we refer the interested reader to the website [1]. Note that the user of AIOCJ does not need to master all the technicalities we discuss here, since they are embedded within AIOCJ. In particular, DPOCs and the projection are automatically handled and hidden from the user.

Proofs, additional details, and examples are available in the companion technical report [9].

2 Dynamic Interaction-Oriented Choreography (DIOC)

This section defines the syntax and semantics of the DIOC language.

The languages that we propose rely on a set *Roles*, ranged over by $r, s, \dots$, whose elements identify the participants in the choreography. Roles exchange messages over channels, also called *operations*: *public operations*, ranged over by o, and *private operations*, ranged over by o^*. We use $o^?$ to range over both public and private operations. Public operations represent relevant communications inside the application. We ensure that both the DIOC and the corresponding DPOC perform the same public operations, in the same order. Vice versa, private communications are used when moving from the DIOC level to the DPOC level, for synchronisation purposes. We denote with *Expr* the set of expressions, ranged over by e. We deliberately do not give a formal definition of expressions and of their typing, since our results do not depend on it. We only require that expressions include at least values, belonging to a set *Val* ranged over by v, and variables, belonging to a set *Var* ranged over by $x, y, \dots$. We also assume a set of boolean expressions ranged over by b.

The syntax of DIOC *processes*, ranged over by $\mathcal{I}, \mathcal{I}', \dots$, is defined as follows:

$$\mathcal{I} ::= o^? : r_1(e) \to r_2(x) \mid \mathcal{I}; \mathcal{I}' \mid \mathcal{I}|\mathcal{I}' \mid x@r = e \mid \mathbf{1} \mid \mathbf{0} \mid$$
$$\texttt{if } b@r \; \{\mathcal{I}\} \; \texttt{else} \; \{\mathcal{I}'\} \mid \texttt{while } b@r \; \{\mathcal{I}\} \mid \texttt{scope } @r \; \{\mathcal{I}\}$$

3

Interaction $o^? : r_1(e) \to r_2(x)$ means that role r_1 sends a message on operation $o^?$ to role r_2 (we require $r_1 \neq r_2$). The sent value is obtained by evaluating expression e in the local state of r_1 and it is then stored in variable x in r_2. Processes $\mathcal{I};\mathcal{I}'$ and $\mathcal{I}|\mathcal{I}'$ denote sequential and parallel composition. Assignment $x@r = e$ assigns the evaluation of expression e in the local state of r to its local variable x. The empty process $\mathbf{1}$ defines a DIOC that can only terminate. $\mathbf{0}$ represents a terminated DIOC. It is needed for the definition of the operational semantics and it is not intended to be used by the programmer. We call *initial* a DIOC process where $\mathbf{0}$ never occurs. Conditional `if` $b@r$ $\{\mathcal{I}\}$ `else` $\{\mathcal{I}'\}$ and iteration `while` $b@r$ $\{\mathcal{I}\}$ are guarded by the evaluation of boolean expression b in the local state of r. The construct `scope @`r $\{\mathcal{I}\}$ delimits a subterm $\mathcal{I}$ of the DIOC process that may be updated in the future. In `scope @`r $\{\mathcal{I}\}$, role r coordinates the updating procedure by interacting with the other roles involved in the scope.

DIOC processes do not execute in isolation: they are equipped with a *global state* Σ and a set of (available) updates $\mathbf{I}$. A global state Σ is a map that defines the value v of each variable x in a given role r, namely $\Sigma : Roles \times Var \to Val$. The local state of role r is $\Sigma_r : Var \to Val$ and it verifies $\forall x \in Var : \Sigma(r, x) = \Sigma_r(x)$. Expressions are always evaluated by a given role r: we denote the evaluation of expression e in local state Σ_r as $[\![e]\!]_{\Sigma_r}$. We assume $[\![e]\!]_{\Sigma_r}$ is always defined (e.g., an error value is given as a result if evaluation is not possible) and that for each boolean expression b, $[\![b]\!]_{\Sigma_r}$ is either `true` or `false`. $\mathbf{I}$ denotes a set of updates, i.e., DIOCs that may replace a scope. $\mathbf{I}$ may change at runtime.

Listing 1.1 gives a realistic example of DIOC process where a `buyer` orders a product from a `seller`, paying via a `bank`.

```
1  price_ok@buyer = false; continue@buyer = true;
2  while ( !price_ok and continue )@buyer {
3   b_prod@buyer = getInput();
4   priceReq : buyer( b_prod ) → seller( s_prod );
5   scope @seller {
6      s_price@seller = getPrice( s_prod );
7      offer : seller( s_price ) → buyer( b_price )
8   };
9   price_ok@buyer = getInput();
10  if ( !price_ok )@buyer {
11    continue@buyer = getInput()} };
12 if ( price_ok )@buyer {
13  payReq : seller( payDesc( s_price ) ) → bank( desc );
14  scope @bank {
15     payment_ok@bank = true;
16     pay : buyer( payAuth( b_price ) ) → bank( auth );
17     ... // code for the payment
18  };
19  if ( payment_ok )@bank {
20     confirm : bank( null ) → seller( _ ) |
21     confirm : bank( null ) → buyer( _ )
```

```
22   } else { abort : bank( null ) → buyer( _ ) } }
```
Listing 1.1. DIOC process for Buying Scenario.

Before starting the application by iteratively asking the price of some goods
to the `seller`, the `buyer` at Line 1 initializes its local variables `price_ok` and
`continue`. Then, by using function `getInput` (Line 3) (s)he reads from the local
console the name of the product to buy and, at Line 4, engages in a communi-
cation via operation `priceReq` with the `seller`. The `seller` computes the price
of the product calling the function `getPrice` (Line 6) and, via operation `offer`,
it sends the price to the `buyer` (Line 7), that stores it in a local variable `b_price`.
These last two operations are performed within a scope, allowing this code to
be updated in the future to deal with changing business rules. If the offer is ac-
cepted, the `seller` sends to the `bank` the payment details (Line 13). The `buyer`
then authorises the payment via operation `pay`. We omit the details of the local
execution of the payment at the `bank`. Since the payment may be critical for
security reasons, the related communication is enclosed in a scope (Lines 14-18),
thus allowing the introduction of a more refined procedure later on. After the
scope successfully terminates, the application ends with the `bank` acknowledging
the payment to the `seller` and the `buyer` in parallel (Lines 20-21). If the pay-
ment is not successful, the failure is notified to the `buyer` only. Note that at Line
1, the annotation `@buyer` means that the variables belong to the `buyer`. Similarly,
at Line 2, the annotation `@buyer` means that the guard of the while is evaluated
by `buyer`. The term `@seller` in Line 5 instead, being part of the scope construct,
indicates the participant that coordinates the code update.

Assume now that the seller direction decides to define new business rules.
For instance, the seller may distribute a fidelity card to buyers, allowing them
to get a 10% discount on their purchases. This business need can be faced by
adding the DIOC below to the set of available updates, so that it can be used to
replace the scope at Lines 5-8 in Listing 1.1.

```
1   cardReq : seller( null ) → buyer( _ );
2   card_id@buyer = getInput();
3   cardRes : buyer( card_id ) → seller( buyer_id );
4   if isValid( buyer_id )@seller {
5     s_price@seller = getPrice( s_prod ) * 0.9
6   } else { s_price@seller = getPrice( s_prod ) };
7   offer : seller( s_price ) → buyer( b_price )
```
Listing 1.2. Fidelity Card Update

When this code executes, the `seller` asks the card ID to the `buyer`. The `buyer`
inputs the ID, stores it into the variable `card_id` and sends this information to
the `seller`. If the card ID is valid then the discount is applied, otherwise the
standard price is computed.

2.1 Connectedness

In order to prove our main result, we require the DIOC code of the updates
and of the starting programs to satisfy a well-formedness syntactic condition

$$\mathsf{transl}(o^? : r_1(e) \to r_2(x)) = \mathsf{transF}(o^? : r_1(e) \to r_2(x)) = \{r_1 \to r_2\}$$

$$\mathsf{transl}(x@r = e) = \mathsf{transF}(x@r = e) = \{r \to r\}$$

$$\mathsf{transl}(\mathbf{1}) = \mathsf{transl}(\mathbf{0}) = \mathsf{transF}(\mathbf{1}) = \mathsf{transF}(\mathbf{0}) = \emptyset$$

$$\mathsf{transl}(\mathcal{I}|\mathcal{I}') = \mathsf{transl}(\mathcal{I}) \cup \mathsf{transl}(\mathcal{I}') \qquad\qquad \mathsf{transF}(\mathcal{I}|\mathcal{I}') = \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}')$$

$$\mathsf{transl}(\mathcal{I};\mathcal{I}') = \begin{cases} \mathsf{transl}(\mathcal{I}') & \text{if } \mathsf{transl}(\mathcal{I}) = \emptyset \\ \mathsf{transl}(\mathcal{I}) & \text{otherwise} \end{cases} \qquad \mathsf{transF}(\mathcal{I};\mathcal{I}') = \begin{cases} \mathsf{transF}(\mathcal{I}) & \text{if } \mathsf{transF}(\mathcal{I}') = \emptyset \\ \mathsf{transF}(\mathcal{I}') & \text{otherwise} \end{cases}$$

$$\mathsf{transl}(\mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) = \mathsf{transl}(\mathtt{while}\ b@r\ \{\mathcal{I}\}) = \{r \to r\}$$

$$\mathsf{transF}(\mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) = \begin{cases} \{r \to r\} & \text{if } \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}') = \emptyset \\ \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}') & \text{otherwise} \end{cases}$$

$$\mathsf{transF}(\mathtt{while}\ b@r\ \{\mathcal{I}\}) = \begin{cases} \{r \to r\} & \text{if } \mathsf{transF}(\mathcal{I}) = \emptyset \\ \mathsf{transF}(\mathcal{I}) & \text{otherwise} \end{cases}$$

$$\mathsf{transl}(\mathtt{scope}\ @r\ \{\mathcal{I}\}) = \{r \to r\}$$

$$\mathsf{transF}(\mathtt{scope}\ @r\ \{\mathcal{I}\}) = \begin{cases} \{r \to r\} & \text{if } \mathsf{roles}(\mathcal{I}) \subseteq \{r\} \\ \bigcup_{r' \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{r\}} \{r' \to r\} & \text{otherwise} \end{cases}$$

Table 1. Auxiliary functions transl and transF.

called *connectedness*. This condition is composed by *connectedness for sequence* and *connectedness for parallel*. Intuitively, connectedness for sequence ensures that the DPOC network obtained by projecting a sequence $\mathcal{I};\mathcal{I}'$ executes first the actions in $\mathcal{I}$ and then those in $\mathcal{I}'$, thus respecting the intended semantics of sequential composition. Connectedness for parallel prevents interferences between parallel interactions. To formally define connectedness we introduce, in Table 1, the auxiliary functions transl and transF that, given a DIOC process, compute sets of pairs representing senders and receivers of possible initial and final interactions in its execution. We represent one such pair as $r_1 \to r_2$. Actions located at r are represented as $r \to r$. For instance, given an interaction $o^? : r_1(e) \to r_2(x)$ both its transl and transF are $\{r_1 \to r_2\}$. For conditional, $\mathsf{transl}(\mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}) = \{r \to r\}$ since the first action executed is the evaluation of the guard by role r. The set $\mathsf{transF}(\mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\})$ is normally $\mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}')$, since the execution terminates with an action from one of the branches. If instead the branches are both empty then transF is $\{r \to r\}$, representing guard evaluation.

We assume a function $\mathsf{roles}(\mathcal{I})$ that computes the roles of a DIOC process $\mathcal{I}$. We also assume a function sig that given a DIOC process returns the set of signatures of its interactions, where the signature of interaction $o^? : r_1(e) \to r_2(x)$ is $o^? : r_1 \to r_2$. For a formal definition of the functions roles and sig we refer the reader to the companion technical report [9].

Definition 1 (Connectedness). *A* DIOC *process $\mathcal{I}$ is connected if it satisfies:*

- **connectedness for sequence:** *each subterm of the form $\mathcal{I}';\mathcal{I}''$ satisfies* $\forall r_1 \to r_2 \in \mathsf{transF}(\mathcal{I}'), \forall s_1 \to s_2 \in \mathsf{transl}(\mathcal{I}'') . \{r_1, r_2\} \cap \{s_1, s_2\} \neq \emptyset;$
- **connectedness for parallel:** *each subterm of the form $\mathcal{I}'|\mathcal{I}''$ satisfies* $\mathsf{sig}(\mathcal{I}') \cap \mathsf{sig}(\mathcal{I}'') = \emptyset.$

[Interaction]

$$\frac{\llbracket e \rrbracket_{\Sigma_{r_1}} = v}{\left\langle A, o^? : r_1(e) \to r_2(x) \right\rangle \xrightarrow{o^? : r_1(v) \to r_2(x)} \left\langle A, x@r_2 = v \right\rangle}$$

[Assign]

$$\frac{\llbracket e \rrbracket_{\Sigma_r} = v}{\left\langle \Sigma, \mathbf{I}, x@r = e \right\rangle \xrightarrow{\tau} \left\langle \Sigma[v/x, r], \mathbf{I}, \mathbf{1} \right\rangle}$$

[Up]

$$\frac{\mathsf{roles}(\mathcal{I}') \subseteq \mathsf{roles}(\mathcal{I}) \quad \mathcal{I}' \in \mathbf{I} \quad \mathcal{I}' \text{ connected}}{\left\langle A, \texttt{scope } @r \ \{\mathcal{I}\} \right\rangle \xrightarrow{\mathcal{I}'} \left\langle A, \mathcal{I}' \right\rangle}$$

[End]

$$\left\langle A, \mathbf{1} \right\rangle \xrightarrow{\sqrt{}} \left\langle A, \mathbf{0} \right\rangle$$

[Sequence]

$$\frac{\left\langle A, \mathcal{I} \right\rangle \xrightarrow{\mu} \left\langle A', \mathcal{I}' \right\rangle \mu \neq \sqrt{}}{\left\langle A, \mathcal{I}; \mathcal{J} \right\rangle \xrightarrow{\mu} \left\langle A', \mathcal{I}'; \mathcal{J} \right\rangle}$$

[Seq-end]

$$\frac{\left\langle A, \mathcal{I} \right\rangle \xrightarrow{\sqrt{}} \left\langle A, \mathcal{I}' \right\rangle \left\langle A, \mathcal{J} \right\rangle \xrightarrow{\mu} \left\langle A, \mathcal{J}' \right\rangle}{\left\langle A, \mathcal{I}; \mathcal{J} \right\rangle \xrightarrow{\mu} \left\langle A, \mathcal{J}' \right\rangle}$$

[NoUp]

$$\left\langle A, \texttt{scope } @r \ \{\mathcal{I}\} \right\rangle \xrightarrow{\text{no-up}} \left\langle A, \mathcal{I} \right\rangle$$

[Change-Updates]

$$\left\langle \Sigma, \mathbf{I}, \mathcal{I} \right\rangle \xrightarrow{\mathbf{I}'} \left\langle \Sigma, \mathbf{I}', \mathcal{I} \right\rangle$$

Table 2. DIOC system semantics (excerpt).

Requiring connectedness does not hamper programmability, since it naturally holds in most of the cases (see, e.g., [1, 8]), and it can always be enforced automatically restructuring the DIOC while preserving its behaviour, following the lines of [16]. Also, connectedness can be checked efficiently.

Theorem 1 (Connectedness-check complexity).
The connectedness of a DIOC *process* $\mathcal{I}$ *can be checked in time* $O(n^2 \log(n))$, *where n is the number of nodes in the abstract syntax tree of* $\mathcal{I}$.

Note that we allow only connected updates. Indeed, replacing a scope with a connected update always results in a deadlock- and race-free DIOC. Thus, there is no need to perform expensive runtime checks to ensure connectedness of the application after an arbitrary sequence of updates has been applied.

2.2 DIOC semantics

We can now define DIOC systems and their semantics.

Definition 2 (DIOC systems). *A* DIOC *system is a triple* $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ *denoting a* DIOC *process* $\mathcal{I}$ *equipped with a global state* Σ *and a set of updates* $\mathbf{I}$.

Definition 3 (DIOC systems semantics). *The semantics of* DIOC *systems is defined as the smallest labelled transition system (LTS) closed under the rules in Table 6 in the companion technical report [9] (excerpt in Table 2), where symmetric rules for parallel composition have been omitted.*

The rules in Table 2 describe the behaviour of a DIOC system by induction on the structure of its DIOC process. We use μ to range over labels. Also, we use A as an abbreviation for $\Sigma, \mathbf{I}$. We comment below on the main rules.

7

Rule [INTERACTION] executes a communication from r_1 to r_2 on operation $o^?$, where r_1 sends to r_2 the value v of an expression e. The value v is then stored in x by r_2. Rule [ASSIGN] evaluates the expression e in the local state Σ_r and stores the resulting value v in the local variable x in role r ($[v/x, r]$ represents the substitution). The rules [UP] and [NOUP] deal with the code replacement and thus the application of an update. Rule [UP] models the application of the update $\mathcal{I}'$ to the scope $\texttt{scope } @r \; \{\mathcal{I}\}$ which, as a result, is replaced by the DIOC process $\mathcal{I}'$. This rule requires the update to be connected. Rule [NOUP] removes the scope boundaries and starts the execution of the body of the scope. Rule [CHANGE-UPDATES] allows the set $\mathbf{I}$ of available updates to change. This rule is always enabled since its execution can happen at any time and the application cannot forbid it.

In our theory, whether to update a scope or not, and which update to apply if many are available, is completely non-deterministic. We have adopted this view to maximize generality. However, for practical applications, one needs rules and conditions which define when an update has to be performed. Refining the semantics to introduce rules for decreasing (or eliminating) the non-determinism would not affect the correctness of our approach. One such refinement has been explored in [8].

We define DIOC *traces*, where all the performed actions are observed, and *weak* DIOC *traces*, where interactions on private operations and silent actions τ are not visible.

Definition 4 (DIOC traces)**.** *A (strong) trace of a* DIOC *system* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{I}_1 \rangle$ *is a sequence (finite or infinite) of labels* $\mu_1, \mu_2, \ldots$ *such that there is a sequence of* DIOC *system transitions* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{I}_1 \rangle \xrightarrow{\mu_1} \langle \Sigma_2, \mathbf{I}_2, \mathcal{I}_2 \rangle \xrightarrow{\mu_2} \ldots$.
A weak trace of a DIOC *system* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{I}_1 \rangle$ *is a sequence of labels* $\mu_1, \mu_2, \ldots$ *obtained by removing all the labels corresponding to private communications, i.e., of the form* $o^* : r_1(v) \to r_2(x)$*, and the silent labels* τ *from a trace of* $\langle \Sigma_1, \mathbf{I}_1, \mathcal{I}_1 \rangle$*.*

3 Dynamic Process-Oriented Choreography (DPOC)

This section describes the syntax and operational semantics of DPOCs. DPOCs include *processes*, ranged over by P, P', …, describing the behaviour of participants. $(P, \Gamma)_r$ denotes a DPOC *role* named r, executing process P in a local state Γ. *Networks*, ranged over by $\mathcal{N}, \mathcal{N}'$, …, are parallel compositions of DPOC roles with different names. DPOC systems, ranged over by $\mathcal{S}$, are DPOC networks equipped with a set of updates $\mathbf{I}$, namely pairs $\langle \mathbf{I}, \mathcal{N} \rangle$.

$$P ::= o^? : x \; \texttt{from } r \mid o^? : e \; \texttt{to } r \mid o^* : X \; \texttt{to } r \mid P; P' \mid P|P' \mid x = e \mid \texttt{while } b \; \{P\}$$
$$\mid \texttt{if } b \; \{P\} \; \texttt{else } \{P'\} \mid n : \texttt{scope } @r \; \{P\} \; \texttt{roles } \{S\} \mid n : \texttt{scope } @r \; \{P\} \mid \mathbf{1} \mid \mathbf{0}$$

$$X ::= \texttt{no} \mid P \qquad\qquad \mathcal{N} ::= (P, \Gamma)_r \mid \mathcal{N} \parallel \mathcal{N}' \qquad\qquad \mathcal{S} ::= \langle \mathbf{I}, \mathcal{N} \rangle$$

Processes include receive action $o^?: x$ `from` r on a specific operation $o^?$ (either public or private) of a message from role r to be stored in variable x, send action $o^?: e$ `to` r of an expression e to be sent to role r, and higher-order send action $o^*: X$ `to` r of the higher-order argument X to be sent to role r. Here X may be either a DPOC process P, which is the new code for a scope in r, or a token `no`, notifying that no update is needed. $P; P'$ and $P|P'$ denote the sequential and parallel composition of P and P', respectively. Processes also feature assignment $x = e$ of expression e to variable x, the process $\mathbf{1}$, that can only successfully terminate, and the terminated process $\mathbf{0}$. We also have conditionals `if` b $\{P\}$ `else` $\{P'\}$ and loops `while` b $\{P\}$. Finally, we have two constructs for scopes. Scope $n:$ `scope @`r $\{P\}$ `roles` $\{S\}$ may occur only inside role r and acts as coordinator to apply (or not apply) the update. The shorter version $n:$ `scope @`r $\{P\}$ is used instead when the role is not the coordinator of the scope. In fact, only the coordinator needs to know the set S of involved roles to communicate which update to apply. Note that scopes are prefixed by an index n. Indexes are unique in each role and are used to avoid interference between different scopes in the same role.

3.1 Projection

Before defining the semantics of DPOCs, we define the projection of a DIOC process onto DPOC processes. This is needed to define the semantics of updates at the DPOC level. The projection exploits auxiliary communications to coordinate the different roles, e.g., ensuring that in a conditional they all select the same branch. To define these auxiliary communications and avoid interference, it is convenient to annotate DIOC main constructs with unique indexes.

Definition 5 (Well-annotated DIOC). *Annotated DIOC processes are obtained by indexing every interaction, assignment, scope, and if and while constructs in a DIOC process with a natural number $n \in \mathbb{N}$, resulting in the following grammar:*

$$\mathcal{I} ::= n : o^? : r_1(e) \to r_2(x) \mid \mathcal{I}; \mathcal{I}' \mid \mathcal{I}|\mathcal{I}' \mid \mathbf{1} \mid \mathbf{0} \mid n : x@r = e$$
$$\mid n : \texttt{while } b@r \ \{\mathcal{I}\} \mid n : \texttt{if } b@r \ \{\mathcal{I}\} \texttt{ else } \{\mathcal{I}'\} \mid n : \texttt{scope @}r \ \{\mathcal{I}\}$$

A DIOC process is well-annotated if all its indexes are distinct.

Note that we can always annotate a DIOC process to make it well-annotated.

We now define the *process-projection function* that derives DPOC processes from DIOC processes. Given an annotated DIOC process $\mathcal{I}$ and a role s, the projected DPOC process $\pi(\mathcal{I}, s)$ is defined by structural induction on $\mathcal{I}$ in Table 3. Here, with a little abuse of notation, we write $\mathsf{roles}(\mathcal{I}, \mathcal{I}')$ for $\mathsf{roles}(\mathcal{I}) \cup \mathsf{roles}(\mathcal{I}')$. We assume that operations o^*_n and variables x_n are never used in the projected DIOC and we use them for auxiliary synchronisations. In most of the cases the projection is trivial. For instance, the projection of an interaction is an output on the sender role, an input on the receiver, and $\mathbf{1}$ on any other role. For a conditional $n:$ `if` $b@r$ $\{\mathcal{I}\}$ `else` $\{\mathcal{I}'\}$, role r locally evaluates the guard and then sends its value to the other roles using auxiliary communications. Similarly, in

$$\boxed{\pi(\mathbf{1}, s)} = \mathbf{1} \qquad \boxed{\pi(\mathbf{0}, s)} = \mathbf{0}$$

$$\boxed{\pi(\mathcal{I}; \mathcal{I}', s)} = \pi(\mathcal{I}, s); \pi(\mathcal{I}', s) \qquad \boxed{\pi(n : x@r = e, s)} = \begin{cases} x = e & \text{if } s = r \\ \mathbf{1} & \text{otherwise} \end{cases}$$

$$\boxed{\pi(\mathcal{I}|\mathcal{I}', s)} = \pi(\mathcal{I}, s) \mid \pi(\mathcal{I}', s)$$

$$\boxed{\pi(n : o^? : r_1(e) \to r_2(x), s)} = \begin{cases} o^? : e \text{ to } r_2 & \text{if } s = r_1 \\ o^? : x \text{ from } r_1 & \text{if } s = r_2 \\ \mathbf{1} & \text{otherwise} \end{cases}$$

$$\boxed{\pi(n : \text{if } b@r \; \{\mathcal{I}\} \text{ else } \{\mathcal{I}'\}, s)} =$$

$$\begin{cases} \text{if } b \; \{(\Pi_{r' \in \text{roles}(\mathcal{I}, \mathcal{I}') \smallsetminus \{r\}} \; o_n^* : true \text{ to } r'); \; \pi(\mathcal{I}, s)\} \\ \qquad \text{else } \{(\Pi_{r' \in \text{roles}(\mathcal{I}, \mathcal{I}') \smallsetminus \{r\}} \; o_n^* : false \text{ to } r'); \; \pi(\mathcal{I}', s)\} & \text{if } s = r \\ o_n^* : x_n \text{ from } r; \text{ if } x_n \; \{\pi(\mathcal{I}, s)\} \text{ else } \{\pi(\mathcal{I}', s)\} & \text{if } r \in \text{roles}(\mathcal{I}, \mathcal{I}') \smallsetminus \{s\} \\ \mathbf{1} & \text{otherwise} \end{cases}$$

$$\boxed{\pi(n : \text{while } b@r \; \{\mathcal{I}\}, s)} =$$

$$\begin{cases} \text{while } b \; \{(\Pi_{r' \in \text{roles}(\mathcal{I}) \smallsetminus \{r\}} o_n^* : true \text{ to } r'); \pi(\mathcal{I}, s); \\ \qquad \Pi_{r' \in \text{roles}(\mathcal{I}) \smallsetminus \{r\}} \; o_n^* : \_ \text{ from } r'\}; & \text{if } s = r \\ \qquad \Pi_{r' \in \text{roles}(\mathcal{I}) \smallsetminus \{r\}} \; o_n^* : false \text{ to } r' \\ o_n^* : x_n \text{ from } r; \\ \qquad \text{while } x_n \; \{\pi(\mathcal{I}, s); \; o_n^* : ok \text{ to } r; o_n^* : x_n \text{ from } r\} & \text{if } s \in \text{roles}(\mathcal{I}) \smallsetminus \{r\} \\ \mathbf{1} & \text{otherwise} \end{cases}$$

$$\boxed{\pi(n : \text{scope } @r \; \{\mathcal{I}\}, s)} = \begin{cases} n : \text{scope } @r \; \{\pi(\mathcal{I}, s)\} \text{ roles } \{\text{roles}(\mathcal{I})\} & \text{if } s = r \\ n : \text{scope } @r \; \{\pi(\mathcal{I}, s)\} & \text{if } s \in \text{roles}(\mathcal{I}) \smallsetminus \{r\} \\ \mathbf{1} & \text{otherwise} \end{cases}$$

Table 3. Process-projection function π.

a loop $n : \text{while } b@r \; \{\mathcal{I}\}$ role r communicates the evaluation of the guard to the other roles. Also, after an iteration has terminated, role r waits for the other roles to terminate and then starts a new iteration. In both the conditional and the loop, indexes are used to choose names for auxiliary operations: the choice is coherent among the different roles and interference between different loops or conditionals is avoided.

There is a trade-off between efficiency and ease of programming that concerns how to ensure that all the roles are aware of the evolution of the computation. Indeed, this can be done in three ways: by using auxiliary communications generated either *i*) by the projection (e.g., as for if and while constructs above) or *ii*) by the semantics (as we will show for scopes) or *iii*) by restricting the class of allowed DIOCs (as done for sequential composition using connectedness for sequence). For instance, auxiliary communications for the $\text{if } b@r \; \{\mathcal{I}\} \text{ else } \{\mathcal{I}'\}$ construct are needed unless one requires that $r \in \{r_1, r_2\}$ for each $r_1 \to r_2 \in \text{transl}(\mathcal{I}) \cup \text{transl}(\mathcal{I}')$. The use of auxiliary communications is

[ONE]
$(\mathbf{1}, \Gamma)_r \xrightarrow{\checkmark} (\mathbf{0}, \Gamma)_r$

[ASSIGN]
$$\frac{[\![e]\!]_\Gamma = v}{(x = e, \Gamma)_r \xrightarrow{\tau} (\mathbf{1}, \Gamma[v/x])_r}$$

[OUT-UP]
$(o^? : X \text{ to } r', \Gamma)_r \xrightarrow{\overline{o^?\langle X\rangle}@r':r} (\mathbf{1}, \Gamma)_r$

[IN]
$(o^? : x \text{ from } r', \Gamma)_r \xrightarrow{o^?(x\leftarrow v)@r':r} (x = v, \Gamma)_r$

[OUT]
$$\frac{[\![e]\!]_\Gamma = v}{(o^? : e \text{ to } r', \Gamma)_r \xrightarrow{\overline{o^?\langle v\rangle}@r':r} (\mathbf{1}, \Gamma)_r}$$

[SEQUENCE]
$$\frac{(P, \Gamma)_r \xrightarrow{\delta} (P', \Gamma')_r \quad \delta \neq \checkmark}{(P;Q, \Gamma)_r \xrightarrow{\delta} (P';Q, \Gamma')_r}$$

[SEQ-END]
$$\frac{(P, \Gamma)_r \xrightarrow{\checkmark} (P', \Gamma)_r \quad (Q, \Gamma)_r \xrightarrow{\delta} (Q', \Gamma')_r}{(P;Q, \Gamma)_r \xrightarrow{\delta} (Q', \Gamma')_r}$$

[LEAD-UP]
$$\frac{\mathcal{I}' = \mathsf{freshIndex}(\mathcal{I}, n) \qquad \mathsf{roles}(\mathcal{I}') \subseteq S}{(n : \mathtt{scope}\ @r\ \{P\}\ \mathtt{roles}\ \{S\}, \Gamma)_r \xrightarrow{\mathcal{I}} (\Pi_{r_i \in S\setminus\{r\}} o_n^* : \pi(\mathcal{I}', r_i)\ \mathtt{to}\ r_i; \pi(\mathcal{I}', r); \Pi_{r_i \in S\setminus\{r\}} o_n^* : \_\ \mathtt{from}\ r_i, \Gamma)_r}$$

[LEAD-NOUP]
$(n : \mathtt{scope}\ @r\ \{P\}\ \mathtt{roles}\ \{S\}, \Gamma)_r \xrightarrow{\ \mathsf{no\text{-}up}\ } (\Pi_{r_i \in S\setminus\{r\}} o_n^* : \mathtt{no\ to}\ r_i; P; \Pi_{r_i \in S\setminus\{r\}} o_n^* : \_\ \mathtt{from}\ r_i, \Gamma)_r$

[UP]
$(n : \mathtt{scope}\ @r'\ \{P\}, \Gamma)_r \xrightarrow{o_n^*(\_\leftarrow P')@r'} (P'; o_n^* : \mathtt{ok\ to}\ r', \Gamma)_r$

[NOUP]
$(n : \mathtt{scope}\ @r'\ \{P\}, \Gamma)_r \xrightarrow{o_n^*(\_\leftarrow \mathrm{no})@r'} (P; o_n^* : \mathtt{ok\ to}\ r', \Gamma)_r$

Table 4. DPOC role semantics (excerpt).

possibly less efficient, while stricter connectedness conditions leave more burden on the shoulders of the programmer.

We now define the projection $\mathsf{proj}(\mathcal{I}, \Sigma)$, based on the process-projection π, to derive a DPOC network from a DIOC process $\mathcal{I}$ and a global state Σ. We denote with $\|_{i \in I} \mathcal{N}_i$ the parallel composition of networks $\mathcal{N}_i$ for each $i \in I$.

Definition 6 (Projection). *The projection of a* DIOC *process $\mathcal{I}$ with global state Σ is the* DPOC *network defined by* $\mathsf{proj}(\mathcal{I}, \Sigma) = \|_{s \in \mathsf{roles}(\mathcal{I})} (\pi(\mathcal{I}, s), \Sigma_s)_s$

The technical report [9] shows the DPOC processes obtained by projecting the DIOC for the Buying scenario in Listing 1.1 on `buyer`, `seller`, and `bank`.

3.2 DPOC semantics

Definition 7 (DPOC systems semantics). *The semantics of* DPOC *systems is defined as the smallest LTS closed under the rules in Table 5 here and Table 7 in the companion technical report [9] (excerpt in Table 4). Symmetric rules for parallel composition have been omitted.*

[LIFT]
$$\frac{\mathcal{N} \xrightarrow{\delta} \mathcal{N}' \quad \delta \neq \mathcal{I}}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\delta} \langle \mathbf{I}, \mathcal{N}' \rangle}$$

[LIFT-UP]
$$\frac{\mathcal{N} \xrightarrow{\mathcal{I}} \mathcal{N}' \quad \mathcal{I} \text{ connected} \quad \mathcal{I} \in \mathbf{I}}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\mathcal{I}} \langle \mathbf{I}, \mathcal{N}' \rangle}$$

[CHANGE-UPDATES]
$$\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\mathbf{I}'} \langle \mathbf{I}', \mathcal{N} \rangle$$

[SYNCH]
$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\overline{o^?}\langle v \rangle @ r_2:r_1} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{o^?(x \leftarrow v) @ r_1:r_2} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{o^?:r_1(v) \rightarrow r_2(x)} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle}$$

[SYNCH-UP]
$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\overline{o^?}\langle X \rangle @ r_2:r_1} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{o^?(\_ \leftarrow X) @ r_1:r_2} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{o^?:r_1(X) \rightarrow r_2(\_)} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle}$$

[EXT-PARALLEL]
$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \eta \neq \sqrt{}}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}'' \rangle}$$

[EXT-PAR-END]
$$\frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\sqrt{}} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{\sqrt{}} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{\sqrt{}} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle}$$

Table 5. DPOC system semantics.

We use δ to range over labels. The semantics in the early style. We comment below on the main rules.

Rule [IN] receives a value v from role r' and assigns it to local variable x of r. Rules [OUT] and [OUT-UP] execute send and higher-order send actions, respectively. The send evaluates expression e in the local state Γ. In rule [ASSIGN], $[v/x]$ represents the substitution of value v for variable x.

Rule [LEAD-UP] concerns the role r coordinating the update of a scope. Role r decides which update to use. It is important that this decision is taken by the unique coordinator r for two reasons. First, r ensures that all involved roles agree on whether to update or not. Second, since the set of updates may change at any time, the choice of the update inside $\mathbf{I}$ needs to be atomic, and this is guaranteed using a unique coordinator. Role r transforms the DIOC $\mathcal{I}$ into $\mathcal{I}'$ using function freshIndex$(\mathcal{I}, n)$, which produces a copy $\mathcal{I}'$ of $\mathcal{I}$. In $\mathcal{I}'$ the indexes of scopes are fresh, which avoids clashes with indexes already present in the target DPOC. Moreover, to avoid that interactions in the update interfere with (parallel) interactions in the context, freshIndex$(\mathcal{I}, n)$ renames all the operations inside $\mathcal{I}$ by adding to them the index n. To this end we extend the set of operations without changing the semantics. For each operation $o^?$ we define extended operations of the form $n \cdot o^?$. The coordinator r also generates the processes to be executed by the roles in S using the process-projection function π. The processes are sent via higher-order communications only to the roles that have to execute them. Then, r starts its own updated code $\pi(\mathcal{I}', r)$. Finally, auxiliary communications are used to synchronise the end of the execution of the replaced process (here $\_$ denotes a fresh variable to store the synchronisation message ok). The auxiliary communications are needed to ensure that the update is performed in a coordi-

nated way, i.e., the roles agree on when the scope starts and terminates and on whether the update is performed or not.

Rule [LEAD-NOUP] instead defines the behaviour when the coordinator r decides to not update. In this case, r sends a token no to each other involved role, notifying them that no update is applied. End of scope synchronisation is as above. Rules [UP] and [NOUP] define the behaviour of the scopes for the other roles involved in the update. The scope waits for a message from the coordinator. If the content of the message is no, the body of the scope is executed. Otherwise, it is a process P' which is executed instead of the body of the scope.

Table 5 defines the semantics of DPOC systems. We use η to range over DPOC systems labels. Rule [LIFT] and [LIFT-UP] lift roles transitions to the system level. [LIFT-UP] also checks that the update $\mathcal{I}$ is connected. Rule [SYNCH] synchronises a send with the corresponding receive, producing an interaction. Rule [SYNCH-UP] is similar, but it deals with higher-order interactions. The labels of these transitions store the information on the occurred communication: label $o^? : r_1(v) \rightarrow r_2(x)$ denotes an interaction on operation $o^?$ from role r_1 to role r_2 where the value v is sent by r_1 and then stored by r_2 in variable x. Label $o^? : r_1(X) \rightarrow r_2(\_)$ denotes a similar interaction, but concerning a higher-order value X. No receiver variable is specified, since the received value becomes part of the code of the receiving process. Rule [EXT-PARALLEL] allows a network inside a parallel composition to compute. Rule [EXT-PAR-END] synchronises the termination of parallel networks. Finally, rule [CHANGE-UPDATES] allows the set of updates to change arbitrarily.

We can now define DPOC traces.

Definition 8 (DPOC traces). *A (strong) trace of a DPOC system $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$ is a sequence (finite or infinite) of labels $\eta_1, \eta_2, \ldots$ with $\eta_i \in \{\tau, o^? : r_1(v) \rightarrow r_2(x), \sqrt{}, \mathcal{I}, \mathsf{no\text{-}up}, \mathbf{I}\}$ such that there is a sequence of transitions*
$\langle \mathbf{I}_1, \mathcal{N}_1 \rangle \xrightarrow{\eta_1} \langle \mathbf{I}_2, \mathcal{N}_2 \rangle \xrightarrow{\eta_2} \ldots.$
A weak trace of a DPOC system $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$ is a sequence of labels $\eta_1, \eta_2, \ldots$ obtained by removing all the labels corresponding to private communications, i.e. of the form $o^* : r_1(v) \rightarrow r_2(x)$ or $o^* : r_1(X) \rightarrow r_2(\_)$, and the silent labels τ, from a trace of $\langle \mathbf{I}_1, \mathcal{N}_1 \rangle$. Furthermore, all the extended operations of the form $n \cdot o^?$ are replaced by $o^?$.

Note that DPOC traces do not include send and receive actions. We do this since these actions have no correspondence at the DIOC level, where only whole interactions are allowed.

Appendix D shows a sample execution of the DPOC obtained by projecting the DIOC for the Buying scenario in Listing 1.1.

4 Correctness

In the previous sections we have presented DIOCs, DPOCs, and described how to derive a DPOC from a given DIOC. This section presents the main technical result of the paper, namely the correctness of the projection. Correctness here

13

means that the weak traces of a connected DIOC coincide with the weak traces of the projected DPOC.

Definition 9 (Trace equivalence). *A* DIOC *system* $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ *and a* DPOC *system* $\langle \mathbf{I}, \mathcal{N} \rangle$ *are (weak) trace equivalent iff their sets of (weak) traces coincide.*

Theorem 2 (Correctness). *For each initial, connected* DIOC *process* $\mathcal{I}$*, each state* Σ*, each set of updates* $\mathbf{I}$*, the* DIOC *system* $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ *and the* DPOC *system* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ *are weak trace equivalent.*

Trace-based properties of the DIOC are inherited by the DPOC. Examples include termination (see [9]) and deadlock-freedom.

Definition 10 (Deadlock-freedom). *An internal* DIOC *(resp.* DPOC*) trace is obtained by removing transitions labelled* $\mathbf{I}$ *from a* DIOC *(resp.* DPOC*) trace. A* DIOC *(resp.* DPOC*) system is deadlock-free if all its maximal finite internal traces have* $\sqrt{}$ *as label of the last transition.*

Intuitively, internal traces are needed since labels $\mathbf{I}$ do not correspond to activities of the application and may be executed also after application termination.

By construction initial DIOCs are deadlock-free. Hence:

Corollary 1 (Deadlock-freedom). *For each initial, connected* DIOC $\mathcal{I}$*, state* Σ*, and set of updates* $\mathbf{I}$ *the* DPOC *system* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ *is deadlock-free.*

Moreover, our DIOCs and DPOCs are free from races and orphan messages. A race occurs when the same receive (resp. send) may interact with different sends (resp. receives). In our setting, an orphan message is an enabled send that is never consumed by a receive. Orphan messages are more relevant in asynchronous systems, where a message may be sent, and stay forever in the network, since the corresponding receive operation may never become enabled. However, even in synchronous systems orphan messages should be avoided: the message is not communicated since the receive is not available, hence a desired behaviour of the application never takes place due to synchronization problems.

Trivially, DIOCs avoid races and orphan messages since send and receive are bound together in the same construct. Differently, at the DPOC level, since all receive of the form $o^? : x \; \texttt{from} \; r_1$ in role r_2 may interact with the sends of the form $o^? : e \; \texttt{to} \; r_2$ in role r_1, races may happen. However, thanks to the correctness of the projection, race-freedom holds also for the projected DPOCs.

Corollary 2 (Race-freedom). *For each initial, connected* DIOC $\mathcal{I}$*, state* Σ*, and set of updates* $\mathbf{I}$*, if* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} \langle \mathbf{I}', \mathcal{N} \rangle$*, then in* $\mathcal{N}$ *two sends (resp. receives) cannot interact with the same receive (resp. send).*

As far as orphan messages are concerned, they may appear in infinite DPOC computations since a receive may not become enabled due to an infinite loop. However, as a corollary of trace equivalence, we have that terminating DPOCs are orphan message-free.

Corollary 3 (Orphan message-freedom). *For each initial, connected* DIOC $\mathcal{I}$*, state* Σ*, and set of updates* $\mathbf{I}$*, if* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \xrightarrow{\mu_1} \cdots \xrightarrow{\sqrt{}} \langle \mathbf{I}', \mathcal{N} \rangle$*, then* $\mathcal{N}$ *contains no sends.*

5 Related works and discussion

This paper presents an approach for the dynamic update of distributed applications. Its distinctive trait is that it guarantees the absence of communication deadlocks and races by construction for the running distributed application, even in presence of updates that were unknown when the application was started. More generally, the DPOC is compliant with the DIOC description, and inherits its properties.

The two approaches closest to ours we are aware of are in the area of multiparty session types [4–6, 13], and deal with dynamic software updates [2] and with monitoring of self-adaptive systems [7]. The main difference between [2] and our approach is that [2] targets concurrent applications which are not distributed. Indeed, it relies on a check on the global state of the application to ensure that the update is safe. Such a check cannot be done by a single role, thus is impractical in a distributed setting. Furthermore, the language in [2] is much more constrained than ours, e.g., requiring each pair of participants to interact on a dedicated pair of channels, and assuming that all the roles not involved in a choice behave the same in the two branches. The approach in [7] is very different from ours, too. In particular, in [7] all the possible behaviours are available since the very beginning, both at the level of types and of processes, and a fixed adaptation function is used to switch between them. This difference derives from the distinction between self-adaptive applications, as they discuss, and applications updated from the outside, as in our case.

We also recall [10], which uses types to ensure safe adaptation. However, [10] allows updates only when no session is active, while we change the behaviour of running DIOCs.

Our work is also similar to [18], which deals with compositionality inside multiparty session types. However, [18] only allows static parallel composition, while we replace a term inside an arbitrary context at runtime.

In principle, our update mechanism can be used to inject guarantees of freedom from deadlocks and races into existing approaches to adaptation, e.g., the ones in the surveys [11, 17]. However, this task is cumbersome, due to the huge number and heterogeneity of those approaches, and since for each of them the integration with our techniques is far from trivial. Nevertheless, we already started it. Indeed, in [8], we apply our technique to the approach described in [14]. While applications in [14] are not distributed and there are no guarantees on the correctness of the application after adaptation, applications in [8], based on the same adaptation mechanisms, are distributed and free from deadlocks and races by construction.

Furthermore, on the website [1], we give examples of how to integrate our approach with distributed [20] and dynamic [23] Aspect-Oriented Programming (AOP) and with Context-Oriented Programming (COP) [12]. In general, we can deal with cross-cutting concerns like logging and authentication, typical of AOP, viewing pointcuts as empty scopes and advices as updates. Layers, typical of COP, can instead be defined by updates which can fire according to contextual conditions. We are also planning to apply our techniques to multiparty session

types [4–6, 13]. The main challenge here is to deal with multiple interleaved sessions. An initial analysis of the problem is presented in [3].

References

1. AIOCJ website. `http://www.cs.unibo.it/projects/jolie/aiocj.html`.
2. G. Anderson and J. Rathke. Dynamic software update for message passing programs. In *APLAS*, volume 7705 of *LNCS*, pages 207–222. Springer, 2012.
3. M. Bravetti et al. Towards global and local types for adaptation. In *SEFM Workshops*, volume 8368 of *LNCS*, pages 3–14. Springer, 2013.
4. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
5. M. Carbone and F. Montesi. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *POPL*, pages 263–274. ACM, 2013.
6. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty session. *Logical Methods in Computer Science*, 8(1), 2012.
7. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-adaptive monitors for multiparty sessions. In *PDP*, pages 688–696. IEEE, 2014.
8. M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli. AIOCJ: A choreographic framework for safe adaptive distributed applications. In *SLE*, volume 8706 of *LNCS*, pages 161–170. Springer, 2014.
9. M. Dalla Preda, I. Lanese, J. Mauro, M. Gabbrielli, and S. Giallorenzo. Dynamic Choreographies: Safe Runtime Updates of Distributed Applications. `http://www.cs.unibo.it/projects/jolie/dioc.pdf`.
10. C. Di Giusto and J. A. Pérez. Disciplined structured communications with consistent runtime adaptation. In *SAC*, pages 1913–1918. ACM, 2013.
11. C. Ghezzi, M. Pradella, and G. Salvaneschi. An evaluation of the adaptation capabilities in programming languages. In *SEAMS*, pages 50–59. ACM, 2011.
12. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM Press, 2008.
14. I. Lanese, A. Bucchiarone, and F. Montesi. A Framework for Rule-Based Dynamic Adaptation. In *TGC*, volume 6084 of *LNCS*, pages 284–300. Springer, 2010.
15. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM*, pages 323–332. IEEE Press, 2008.
16. I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In *WWV*, volume 123, pages 34–48. EPTCS, 2013.
17. L. A. F. Leite et al. A systematic literature review of service choreography adaptation. *Service Oriented Computing and Applications*, 7(3):199–216, 2013.
18. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
19. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.
20. R. Pawlak et al. JAC: an aspect-based distributed dynamic framework. *Softw., Pract. Exper.*, 34(12):1119–1148, 2004.
21. Rust website. `http://www.rust-lang.org/`.
22. Scribble website. `http://www.jboss.org/scribble`.

23. Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS*, pages 85–92. ACM, 2002.

A Rules for the DIOC system semantics

Table 6 presents the rules of the DIOC system semantics (symmetric rules for parallel composition are omitted).

[INTERACTION]
$$\frac{[\![e]\!]_{\Sigma_{r_1}} = v}{\langle A, o^? : r_1(e) \to r_2(x)\rangle \xrightarrow{o^?:r_1(v)\to r_2(x)} \langle A, x@r_2 = v\rangle}$$

[SEQUENCE]
$$\frac{\langle A, \mathcal{I}\rangle \xrightarrow{\mu} \langle A', \mathcal{I}'\rangle\, \mu \neq \surd}{\langle A, \mathcal{I}; \mathcal{J}\rangle \xrightarrow{\mu} \langle A', \mathcal{I}'; \mathcal{J}\rangle}$$

[ASSIGN]
$$\frac{[\![e]\!]_{\Sigma_r} = v}{\langle \Sigma, \mathbf{I}, x@r = e\rangle \xrightarrow{\tau} \langle \Sigma[v/x, r], \mathbf{I}, \mathbf{1}\rangle}$$

[SEQ-END]
$$\frac{\langle A, \mathcal{I}\rangle \xrightarrow{\surd} \langle A, \mathcal{I}'\rangle\, \langle A, \mathcal{J}\rangle \xrightarrow{\mu} \langle A, \mathcal{J}'\rangle}{\langle A, \mathcal{I}; \mathcal{J}\rangle \xrightarrow{\mu} \langle A, \mathcal{J}'\rangle}$$

[PARALLEL]
$$\frac{\langle A, \mathcal{I}\rangle \xrightarrow{\mu} \langle A', \mathcal{I}'\rangle\, \mu \neq \surd}{\langle A, \mathcal{I} \parallel \mathcal{J}\rangle \xrightarrow{\mu} \langle A', \mathcal{I}' \parallel \mathcal{J}\rangle}$$

[PAR-END]
$$\frac{\langle A, \mathcal{I}\rangle \xrightarrow{\surd} \langle A, \mathcal{I}'\rangle\, \langle A, \mathcal{J}\rangle \xrightarrow{\surd} \langle A, \mathcal{J}'\rangle}{\langle A, \mathcal{I} \parallel \mathcal{J}\rangle \xrightarrow{\surd} \langle A, \mathcal{I}' \parallel \mathcal{J}'\rangle}$$

[IF-THEN]
$$\frac{[\![b]\!]_{\Sigma_r} = \mathtt{true}}{\langle A, \mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}\rangle \xrightarrow{\tau} \langle A, \mathcal{I}\rangle}$$

[IF-ELSE]
$$\frac{[\![b]\!]_{\Sigma_r} = \mathtt{false}}{\langle A, \mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}\rangle \xrightarrow{\tau} \langle A, \mathcal{I}'\rangle}$$

[WHILE-UNFOLD]
$$\frac{[\![b]\!]_{\Sigma_r} = \mathtt{true}}{\langle A, \mathtt{while}\ b@r\ \{\mathcal{I}\}\rangle \xrightarrow{\tau} \langle A, \mathcal{I}; \mathtt{while}\ b@r\ \{\mathcal{I}\}\rangle}$$

[WHILE-EXIT]
$$\frac{[\![b]\!]_{\Sigma_r} = \mathtt{false}}{\langle A, \mathtt{while}\ b@r\ \{\mathcal{I}\}\rangle \xrightarrow{\tau} \langle A, \mathbf{1}\rangle}$$

[UP]
$$\frac{\mathsf{roles}(\mathcal{I}') \subseteq \mathsf{roles}(\mathcal{I})\quad \mathcal{I}' \in \mathbf{I}\quad \mathcal{I}'\ \text{connected}}{\langle A, \mathtt{scope}\ @r\ \{\mathcal{I}\}\rangle \xrightarrow{\mathcal{I}'} \langle A, \mathcal{I}'\rangle}$$

[NOUP]
$$\langle A, \mathtt{scope}\ @r\ \{\mathcal{I}\}\rangle \xrightarrow{\mathsf{no\text{-}up}} \langle A, \mathcal{I}\rangle$$

[END]
$$\langle A, \mathbf{1}\rangle \xrightarrow{\surd} \langle A, \mathbf{0}\rangle$$

[CHANGE-UPDATES]
$$\langle \Sigma, \mathbf{I}, \mathcal{I}\rangle \xrightarrow{\mathbf{I}'} \langle \Sigma, \mathbf{I}', \mathcal{I}\rangle$$

Table 6. DIOC system semantics.

We describe here the rules whose descriptions were omitted in Table 2 for space reasons.

Rule [END] terminates the execution of an empty process. Rule [SEQUENCE] executes a step in the first process of a sequential composition, while rule [SEQ-END] acknowledges the termination of the first process, starting the second one. Rule [PARALLEL] allows a process in a parallel composition to compute, while rule [PAR-END] synchronises the termination of two parallel processes. Rules [IF-THEN] and [IF-ELSE] evaluate the boolean guard of a conditional, selecting the then and the else branch, respectively. Rules [WHILE-UNFOLD] and [WHILE-EXIT] correspond respectively to the unfolding of a while when its condition is satisfied and to its termination otherwise.

B   Rules for the DPOC role semantics

Table 7 presents the rules of the DPOC role semantics (symmetric rules for parallel composition are omitted).

[ONE]
$$(\mathbf{1}, \Gamma)_r \xrightarrow{\checkmark} (\mathbf{0}, \Gamma)_r$$

[OUT-UP]
$$(o^? : X \text{ to } r', \Gamma)_r \xrightarrow{\overline{o^?}\langle X \rangle @r':r} (\mathbf{1}, \Gamma)_r$$

[ASSIGN]
$$\frac{\llbracket e \rrbracket_\Gamma = v}{(x = e, \Gamma)_r \xrightarrow{\tau} (\mathbf{1}, \Gamma[v/x])_r}$$

[IN]
$$(o^? : x \text{ from } r', \Gamma)_r \xrightarrow{o^?(x \leftarrow v)@r':r} (x = v, \Gamma)_r$$

[OUT]
$$\frac{\llbracket e \rrbracket_\Gamma = v}{(o^? : e \text{ to } r', \Gamma)_r \xrightarrow{\overline{o^?}\langle v \rangle @r':r} (\mathbf{1}, \Gamma)_r}$$

[SEQUENCE]
$$\frac{(P, \Gamma)_r \xrightarrow{\delta} (P', \Gamma')_r \quad \delta \neq \checkmark}{(P; Q, \Gamma)_r \xrightarrow{\delta} (P'; Q, \Gamma')_r}$$

[SEQ-END]
$$\frac{(P, \Gamma)_r \xrightarrow{\checkmark} (P', \Gamma)_r \quad (Q, \Gamma)_r \xrightarrow{\delta} (Q', \Gamma')_r}{(P; Q, \Gamma)_r \xrightarrow{\delta} (Q', \Gamma')_r}$$

[PARALLEL]
$$\frac{(P, \Gamma)_r \xrightarrow{\delta} (P', \Gamma')_r \quad \delta \neq \checkmark}{(P \mid Q, \Gamma)_r \xrightarrow{\delta} (P' \mid Q, \Gamma')_r}$$

[PAR-END]
$$\frac{(P, \Gamma)_r \xrightarrow{\checkmark} (P', \Gamma)_r \quad (Q, \Gamma)_r \xrightarrow{\checkmark} (Q', \Gamma)_r}{(P \mid Q, \Gamma)_r \xrightarrow{\checkmark} (P' \mid Q', \Gamma)_r}$$

[IF-THEN]
$$\frac{\llbracket b \rrbracket_\Gamma = \text{true}}{(\text{if } b \; \{P\} \text{ else } \{P'\}, \Gamma)_r \xrightarrow{\tau} (P, \Gamma)_r}$$

[IF-ELSE]
$$\frac{\llbracket b \rrbracket_\Gamma = \text{false}}{(\text{if } b \; \{P\} \text{ else } \{P'\}, \Gamma)_r \xrightarrow{\tau} (P', \Gamma)_r}$$

[WHILE-UNFOLD]
$$\frac{\llbracket b \rrbracket_\Gamma = \text{true}}{(\text{while } b \; \{P\}, \Gamma)_r \xrightarrow{\tau} (P; \text{while } e \; \{P\}, \Gamma)_r}$$

[WHILE-EXIT]
$$\frac{\llbracket b \rrbracket_\Gamma = \text{false}}{(\text{while } b \; \{P\}, \Gamma)_r \xrightarrow{\tau} (\mathbf{1}, \Gamma)_r}$$

[LEAD-UP]
$$\frac{\mathcal{I}' = \text{freshIndex}(\mathcal{I}, n) \qquad \text{roles}(\mathcal{I}') \subseteq S}{\begin{array}{l}(n : \text{scope } @r \; \{P\} \text{ roles } \{S\}, \Gamma)_r \xrightarrow{\mathcal{I}} \\ \quad (\Pi_{r_i \in S \setminus \{r\}} o_n^* : \pi(\mathcal{I}', r_i) \text{ to } r_i; \pi(\mathcal{I}', r); \Pi_{r_i \in S \setminus \{r\}} o_n^* : \_ \text{ from } r_i, \Gamma)_r\end{array}}$$

[LEAD-NOUP]
$$\begin{array}{l}(n : \text{scope } @r \; \{P\} \text{ roles } \{S\}, \Gamma)_r \xrightarrow{\text{no-up}} \\ \quad (\Pi_{r_i \in S \setminus \{r\}} o_n^* : \text{no to } r_i; P; \Pi_{r_i \in S \setminus \{r\}} o_n^* : \_ \text{ from } r_i, \Gamma)_r\end{array}$$

[UP]
$$(n : \text{scope } @r' \; \{P\}, \Gamma)_r \xrightarrow{o_n^*(\_\leftarrow P')@r'} (P'; o_n^* : \text{ok to } r', \Gamma)_r$$

[NOUP]
$$(n : \text{scope } @r' \; \{P\}, \Gamma)_r \xrightarrow{o_n^*(\_\leftarrow\text{no})@r'} (P; o_n^* : \text{ok to } r', \Gamma)_r$$

Table 7. DPOC role semantics.

We describe here the rules whose descriptions were omitted in Table 4 for space reasons.

Rules [SEQUENCE] executes a step in the first process of a sequential composition, while rule [SEQ-END] acknowledges the termination of the first process,

starting the second one. Rules [PARALLEL] allows a process in a parallel composition to execute, while rule [PAR-END] synchronises the termination of two parallel processes. Rules [IF-THEN] and [IF-ELSE] execute the then or the else branch in a conditional, respectively. Rules [WHILE-UNFOLD] and [WHILE-EXIT] model the unfolding or the termination of a loop.

C   Projecting the DIOC for the Buying scenario on `buyer`, `seller`, and `bank`.

This section shows the projections of the DIOC process defined in Listing 1.1 on the `bank`, `buyer`, and `seller` roles.

In order to define the projection we first have to annotate the DIOC. This leads to the following annotated DIOC.

```
1 : price_ok@buyer = false;
2 : continue@buyer = true;
3 : while ( !price_ok and continue )@buyer {
 4 : b_prod@buyer = getInput();
 5 : priceReq : buyer( b_prod ) → seller( s_prod );
 6 : scope @seller {
   7 : s_price@seller = getPrice( s_prod );
   8 : offer : seller( s_price ) → buyer( b_price )
 };
 9: price_ok@buyer = getInput();
 10 : if ( !price_ok )@buyer {
  11 : continue@buyer = getInput()} };
12 : if ( price_ok )@buyer {
 13 : payReq : seller( payDesc( s_price ) ) → bank( desc );
 14 : scope @bank {
   15 : payment_ok@bank = true;
   16 : pay : buyer( payAuth( b_price ) ) → bank( auth );
   ... // code for the payment
 };
 17 : if ( payment_ok )@bank {
   18 : confirm : bank( null ) → seller( _ ) |
   19 : confirm : bank( null ) → buyer( _ )
 } else {
   20 : abort : bank( null ) → buyer( _ )
 } }
```

Listing 1.3. Annotated DIOC process for Buying Scenario.

We are ready to compute the projection on the `bank`, `buyer`, and `seller` roles respectively. To improve readability, we omit some **1** processes that have no impact on the behaviour.

```
o*₁₂ : x₁₂ from buyer;
if ( x₁₂ ) {
  payReq : desc from seller;
  14 : scope @bank {
        payment_ok = true;
        pay : auth from buyer;
        ... // code for the payment
        }
      roles { buyer , bank };
  if ( payment_ok ) {
    { o*₁₇ : true to seller | o*₁₇ : true to buyer };
    { confirm : null to seller | confirm : null to buyer }
  } else {
    { o*₁₇ : true to seller | o*₁₇ : true to buyer };
    abort : null to buyer } }
```

Listing 1.4. Bank DPOC Process

```
price_ok = false; continue = true;
while ( not( price_ok ) and continue ) {
  o*₃ : true to seller;
  b_prod = getInput ();
  priceReq : b_prod to seller;
  6 : scope @seller {
      offer : b_price from seller }
  price_ok = getInput ();
  if ( not( price_ok ) ) { continue = getInput () };
  o*₃ : _ from seller };
o*₃ : false to seller;
if ( price_ok ) {
  { o*₁₂ : true to seller | o*₁₂ : true to bank };
  14 : scope payment@bank {
    pay : payAuth( b_price ) to bank };
  o*₁₇ : x₁₇ from bank;
  if ( x₁₇ ) { confirm : _ from bank
  } else { abort : _ from bank } }
```

Listing 1.5. Buyer DPOC Process

```
o*₃ : x₃ from buyer;
while ( x₃ ) {
  priceReq : s_prod from buyer;
  6 : scope @seller {
        s_price = getPrice( s_prod );
        offer : s_price to buyer }
      roles { seller , buyer };
  o*₃ : ok to buyer;
  o*₃ : x₃ from buyer };
o*₁₂ : x₁₂ from buyer;
if ( x₁₂ ) {
  payReq : payDesc( s_price ) to bank;
  o*₁₇ : x₁₇ from bank;
  if ( x₁₇ ) { confirm : _ from bank } }
```

Listing 1.6. Seller DPOC Process

D   Running example of scope update

This section shows an example of how updates are performed. We consider an excerpt of the choreography of the Buying Scenario (Listing 1.1) simulating the update of the scope in Lines 5-8. To this end, we assume that the seller direction decides to stimulate business by using the update in Listing 1.2.

Let us consider both the DIOC and the DPOC level, dropping some **1**s to improve readability. Assume that the buyer has just sent the name of the product (s)he is interested in to the seller (Line 4) and consider the following annotated DIOC:

```
6 : scope @seller {
  7 : s_price@seller = getPrice( s_prod );
  8 : offer : seller( s_price ) ) → buyer( b_price )
}
```

At the DIOC level, the scope price-inquiry is atomically substituted with the new code with fresh indexes. Then, the DIOC reduces to:

```
21 : cardReq : seller( null ) → buyer( _ );
22 : card_id@buyer = getInput();
23 : card : buyer( card_id ) → seller( buyer_id );
24 : if isValid( buyer_id )@seller {
  25 : s_price@seller = getPrice( s_prod ) * 0.9
} else {
  26 : s_price@seller = getPrice( s_prod )
};
27 : offer : seller( s_price ) → buyer( b_price )
```

At the DPOC level, this operation is not atomic, since the scope is distributed between two participants, and the coordination protocol is explicitly represented.

To clarify this point, let us consider the DPOC process P_b below, obtained by projecting the DIOC of the update in Listing 1.2 on the buyer role.

```
P_b := cardReq : null from seller;
  card_id = getInput();
  card : card_id to seller;
  offer : b_price from seller
```

At the DPOC level, the first step of the update protocol is performed by the seller. The DPOC description of the seller before the update is:

```
6 : scope @seller {
      s_price = getPrice( s_prod );
      offer : s_price to buyer }
      roles { seller, buyer }
```

When the scope construct is enabled, the seller, being the coordinator of the update, decides to update using the code in Listing 1.2. Thus, the seller reduces to:

```
o_6* : P_b to buyer;
cardReq : null to buyer;
card : buyer_id from buyer;
if isValid( buyer_id ) {
  s_price = getPrice( s_prod ) * 0.9
  } else { s_price@seller = getPrice( s_prod ) };
offer : s_price to buyer;
o_6* : _ from buyer;
```

First, the seller requires the buyer to update, sending to him the new DPOC fragment to execute. Then, the seller starts to execute its own updated DPOC. When the new DPOC code is terminated, (s)he waits for the notification of the termination of the DPOC fragment executed by the buyer.

As far as the buyer is concerned, the DPOC before the update is as follows.

```
6 : scope @seller {
  offer : s_price from seller
}
```

The scope construct in the buyer waits for the arrival of a message from the coordinator of the update. In case an update has to be applied, this message contains the DPOC fragment to execute. Once this message is received, the scope construct is replaced by the received DPOC fragment, followed by the notification of termination to the seller.

```
P_b ; o_6* : ok to seller
```

Let us now consider the case where the application is not updated. At the DIOC level, the scope construct simply disappears, and its body becomes enabled.

```
s_price@seller = getPrice( s_prod );
offer : seller( s_price ) ) → buyer( b_price )
```

As before, at the `DPOC` level this operation is not atomic. In particular, the `DPOC` process of the `seller` becomes as follows.

```
o₆* : no to buyer;
s_price = getPrice( s_prod );
offer :  s_price to buyer;
o₆* : _ from buyer;
```

Here the `seller` notifies to the `buyer` that no update is performed, and then proceeds with the normal execution. Then, as before, (s)he waits for the notification of the termination of the body of the scope from the `buyer`. Dually, the `buyer` waits for the arrival of the message. If the message states that no update is needed, the scope construct is removed and its body executed. At the end, a notification of termination is sent to the coordinator of the update:

```
offer :  b_price from seller;
o₆* : ok to seller;
```

E   Proof of Theorem 1

In order to prove the bound on the complexity of the connectedness check we use the lemma below, showing that the checks to verify the connectedness for sequence for a single sequence operator can be performed in linear time on the size of the sets generated by transI and transF.

Lemma 1. *Given S, S' sets of multisets of two elements, checking if $\forall s \in S . \forall s' \in S' . s \cap s' \neq \emptyset$ can be done in $O(n)$ steps, where n is the maximum of $|S|$ and $|S'|$.*

Proof. W.l.o.g. we can assume that $|S| \leq |S'|$. If $|S| \leq 9$ then the check can be performed in $O(n)$ by comparing all the elements in S with all the elements in S'. If $|S| > 9$ then at least 4 distinct elements appear in the multisets in S since the maximum number of multisets with cardinality 2 obtained by 3 distinct elements is 9. In this case the following cases cover all the possibilities:

- there exist distinct elements a, b, c, d s.t. $\{a, b\}, \{a, c\}$, and $\{a, d\}$ belong to S. In this case for the check to succeed all the multisets in S' must contain a, otherwise the intersection of the multiset not containing a with one among the multisets $\{a, b\}, \{a, c\}$, and $\{a, d\}$ is empty. Similarly, since $|S'| > 9$, for the check to succeed all the multisets in S must contain a. Hence, if $\{a, b\}, \{a, c\}$, and $\{a, d\}$ belong to S then the check succeeds iff a belongs to all the multisets in S and in S'.
- there exist distinct elements a, b, c, d s.t. $\{a, b\}$ and $\{c, d\}$ belong to S. In this case the check succeeds only if S' is a subset of $\{\{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}$. Since $|S'| > 9$ the check can never succeed.
- there exist distinct elements a, b, c s.t. $\{a, a\}$ and $\{b, c\}$ belong to S. In this case the check succeeds only if S' is a subset of $\{\{a, b\}, \{a, c\}\}$. Since $|S'| > 9$ the check can never succeed.

24

– there exist distinct elements a, b s.t. $\{a, a\}$ and $\{b, b\}$ belong to S. In this case the check succeeds only if S' is a subset of $\{\{a, b\}\}$. Since $|S'| > 9$ the check can never succeed.

Summarising, if $|S| > 9$ the check can succeed iff all the multisets in S and in S' share a common element. The existence of such an element can be verified in time $O(n)$.

Theorem 1 (Connectedness-check complexity).
The connectedness of a DIOC process $\mathcal{I}$ can be checked in time $O(n^2 \log(n))$, where n is the number of nodes in the abstract syntax tree of $\mathcal{I}$.

Proof. To check the connectedness of $\mathcal{I}$ we first compute the values of the functions transl, transF, and sig for each node of the abstract syntax tree (AST). We then check for each sequence operator whether connectedness for sequence holds and for each parallel operator whether connectedness for parallel holds.

The functions transl and transF associate to each node a set of pairs of roles. Assuming an implementation of the data set structure based on balanced trees (with pointers), transl and transF can be computed in constant time for interactions, assignments, $\mathbf{1}$, $\mathbf{0}$, and sequence constructs. For while and scope constructs computing $\mathsf{transF}(\mathcal{I}')$ requires the creation of balanced trees having an element for every role of $\mathcal{I}'$. Since the roles are $O(n)$, $\mathsf{transF}(\mathcal{I}')$ can be computed in $O(n \log(n))$. For parallel and if constructs a union of sets is needed. The union costs $O(n \log(n))$ since each set generated by transl and transF contains at maximum n elements.

The computation of sig can be performed in $O(1)$ except for the parallel, sequence, and if constructs, where the union of sets costs $O(n \log(n))$. Since the AST contains n nodes, the computation of the sets generated by transl, transF, and sig can be performed in $O(n^2 \log(n))$.

To check connectedness for sequence we have to verify that for each node $\mathcal{I}'; \mathcal{I}''$ of the AST $\forall r_1 \rightarrow r_2 \in \mathsf{transF}(\mathcal{I}'), \forall s_1 \rightarrow s_2 \in \mathsf{transl}(\mathcal{I}'') . \{r_1, r_2\} \cap \{s_1, s_2\} \neq \emptyset$. Since $\mathsf{transF}(\mathcal{I}')$ and $\mathsf{transl}(\mathcal{I}'')$ have $O(n)$ elements, thanks to Lemma 1, checking if $\mathcal{I}'; \mathcal{I}''$ is connected for sequence costs $O(n)$. Since in the AST there are less than n sequence operators, checking the connectedness for sequence on the whole AST costs $O(n^2)$.

To check connectedness for parallel we have to verify that for each node $\mathcal{I}'|\mathcal{I}''$ of the AST we have that $\mathsf{sig}(\mathcal{I}') \cap \mathsf{sig}(\mathcal{I}'') = \emptyset$. Since $\mathsf{sig}(\mathcal{I}')$ and $\mathsf{sig}(\mathcal{I}'')$ have $O(n)$ elements, checking if their intersection is empty costs $O(n \log(n))$. Since in the AST there are less than n parallel operators, checking the connectedness for parallel on the whole AST costs $O(n^2 \log(n))$.

The complexity of checking the connectedness of the entire AST is therefore limited by the cost of computing functions transl, transF, and sig, and of checking the connectedness for parallel. All these activities have a complexity of $O(n^2 \log(n))$.

F   Proof of Theorem 2

This section presents the proof of our main result, Theorem 2, including various auxiliary definitions and lemmas.

The proof strategy consists in defining a notion of bisimilarity (Definition 20) which implies weak trace equivalence (Lemma 9) and then providing a suitable bisimulation relating each well-annotated connected DIOC system with its projection. Such a relation is not trivial, since events which are atomic in the DIOC, e.g., the evaluation of the guard of a conditional (including removing the discarded branch), are no more atomic in the DPOC. In the case of conditional, the DIOC transition is mimicked by a conditional performed by the role evaluating the guard, a set of auxiliary communications sending the value of the guard to the other roles, and local conditionals based on the received value. These mismatches are taken care by function upd (Definition 19). This function needs also to remove the auxiliary communications allowing to synchronise the termination of scopes, which have no counterpart after the DIOC scope has been consumed. However, we have to record their impact on the possible executions. Thus we define an event structure for DIOC (Definition 12) and one for DPOC (Definition 15) and we show that the two are related (Lemma 2).

In the main part, we defined annotated DIOCs (Definition 5). Here we also need to speak about their semantics. Indeed, annotated DIOCs trivially inherit the semantics of DIOCs, since indexes are just decorations, with no effect on the behaviour. The only tricky points are in rule [INTERACTION], where the assignment inherits the index from the interaction, in rule [WHILE-UNFOLD], where the body is copied together with its indexes, and in rule [UP], where one has to ensure that indexes of constructs from the body of the update are never used elsewhere in the DIOC.

Notably, due to while unfolding, uniqueness of indexes is not preserved by transitions. To solve this problem we build global indexes on top of indexes. Uniqueness of global indexes is preserved by transitions. The same construction can be applied both at the DIOC level and at the DPOC level.

Definition 11 (Global index). *Given an annotated* DIOC *process* $\mathcal{I}$, *or an annotated* DPOC *network* $\mathcal{N}$ *(defined later on), for each annotated construct with index n we define its global index ξ as follows:*

- *if the construct is not in the body of a while then $\xi = n$;*
- *if the innermost while construct that contains the considered construct has global index ξ' then the considered construct has global index $\xi = \xi' : n$.*

Using global indexes we can now define event structures corresponding to the execution of DIOCs and DPOCs. We start by defining DIOC events. Some events correspond to transitions of the DIOC, and we say that they are enabled when the corresponding transition is enabled, executed when the corresponding transition is executed. DIOC events are defined on annotated DIOCs. Note that a non-annotated DIOC can always be annotated.

26

Definition 12 (DIOC **events**). *We use ε to range over events, and we write $[\varepsilon]_r$ to highlight that event ε is performed by role r. An annotated* DIOC *$\mathcal{I}$ contains the following events:*

Communication events: *a sending event $\xi : \overline{o^?}@r_2$ in role r_1 and a receiving event $\xi : o^?@r_1$ in role r_2 for each interaction $n : o^? : r_1(e) \to r_2(x)$ with global index ξ; we also denote the sending event as f_ξ or $[f_\xi]_{r_1}$ and the receiving event as t_ξ or $[t_\xi]_{r_2}$. Sending and receiving events correspond to the transition executing the interaction.*

Assignment events: *an assignment event ε_ξ in role r for each assignment $n : x@r = e$ with global index ξ; the event corresponds to the transition executing the assignment.*

Scope events: *a scope initialisation event $\uparrow_\xi$ and a scope termination event $\downarrow_\xi$ for each scope $n : \mathtt{scope}\ @r\ \{\mathcal{I}\}$ with global index ξ. Both these events belong to all the roles in $\mathsf{roles}(\mathcal{I})$. The scope initialisation event corresponds to the transition performing or not performing an update on the given scope. The scope termination event is just an auxiliary event (related to the auxiliary interactions implementing the scope termination).*

If events: *a guard if-event ε_ξ in role r for each construct $n : \mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I'}\}$ with global index ξ; the guard-if event corresponds to the transition evaluating the guard of the if.*

While events: *a guard while-event ε_ξ in role r for each construct $n : \mathtt{while}\ b@r\ \{\mathcal{I}\}$ with global index ξ; the guard-while event corresponds to the transition evaluating the guard of the while.*

Function $\mathsf{events}(\mathcal{I})$ denotes the set of events of the annotated DIOC *$\mathcal{I}$. A sending and a receiving event with the same global index ξ are called matching events. We denote with $\overline{\varepsilon}$ an event matching event ε.*

Note that there are events corresponding to just one execution of the while. If unfolding is performed, new events are created.

The relation below defines a causality relation among events based on the constraints given by the semantics on the execution of the corresponding transitions.

Definition 13 (DIOC **causality relation**). *Let us consider an annotated* DIOC *$\mathcal{I}$. A causality relation $\leq_{\mathsf{DIOC}} \subseteq \mathsf{events}(\mathcal{I}) \times \mathsf{events}(\mathcal{I})$ is a partial order among events in $\mathcal{I}$. We define $\leq_{\mathsf{DIOC}}$ as the minimum partial order satisfying:*

Sequentiality: *let $\mathcal{I'}; \mathcal{I''}$ be a subterm of* DIOC *$\mathcal{I}$. If ε' is an event in $\mathcal{I'}$ and ε'' is an event in $\mathcal{I''}$, then $\varepsilon' \leq_{\mathsf{DIOC}} \varepsilon''$.*

Scope: *let $n : \mathtt{scope}\ @r\ \{\mathcal{I'}\}$ be a subterm of* DIOC *$\mathcal{I}$. If ε' is an event in $\mathcal{I'}$ then $\uparrow_\xi \leq_{\mathsf{DIOC}} \varepsilon' \leq_{\mathsf{DIOC}} \downarrow_\xi$.*

Synchronisation: *for each interaction the sending event precedes the receiving event.*

If: *let $n : \mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I'}\}$ be a subterm of* DIOC *$\mathcal{I}$, let ε_ξ be the guard if-event in role r, then for every event ε in $\mathcal{I}$ and for every event ε' in $\mathcal{I'}$ we have $\varepsilon_\xi \leq_{\mathsf{DIOC}} \varepsilon$ and $\varepsilon_\xi \leq_{\mathsf{DIOC}} \varepsilon'$.*

While: *let $n : \mathtt{while}\ b@r\ \{\mathcal{I}\}$ be a subterm of* DIOC *$\mathcal{I}$, let ε_ξ be the guard while-event in role r, then for every event ε in $\mathcal{I'}$ we have $\varepsilon_\xi \leq_{\mathsf{DIOC}} \varepsilon$.*

We now define events and the corresponding causality relation also for DPOCs. First, we need to define annotated DPOCs. Annotations for DPOCs exploit not only indexes $i \in \mathbb{N}$, but also indexes of the form $(i, true)$ and $(i, false)$ with $i \in \mathbb{N}$. We use n to range over all forms of indexes.

Definition 14 (Annotated DPOC). *In DPOC networks, scopes are already annotated. Annotated DPOC networks are obtained by adding indexes n also to communication primitives, assignments, while, and if constructs, thus obtaining the following grammar:*

$$
\begin{aligned}
P ::=\ & n : o^? : x \ \texttt{from}\ r \mid n : o^? : e \ \texttt{to}\ r \mid n : o^* : X \ \texttt{to}\ r \mid \\
& P; P' \mid\ P \mid P' \mid n : x = e \mid \mathbf{1} \mid \mathbf{0} \mid \\
& n : \texttt{if}\ b\ \{P\}\ \texttt{else}\ \{P'\} \mid n : \texttt{while}\ b\ \{P\} \mid \\
& n : \texttt{scope}\ @r\ \{P\}\ \texttt{roles}\ \{S\} \mid \\
& n : \texttt{scope}\ @r\ \{P\} \\
X ::=\ & \texttt{no} \mid P \\
\mathcal{N} ::=\ & (P, \Gamma)_r \mid \mathcal{N} \parallel \mathcal{N}'
\end{aligned}
$$

We extend the projection function so to generate annotated DPOC networks from annotated DIOC processes. It requires that all the DPOC constructs obtained projecting a DIOC construct with index n have index n with the only exception of the index of the auxiliary communications of the projection of the if and while constructs. In particular, for the projection of the if construct and for each role except the coordinator, we assign to the auxiliary input communications a fresh index i. As far as the coordinator is concerned instead, the auxiliary output communications in the if branch are indexed with $(i, true)$ while the auxiliary communications in the else branch are indexed with $(i, false)$, where i is the fresh index associated to the target role. The indexes of the auxiliary operations of the while construct projection are instead computed as follows:

- for each non coordinator role we choose a pair of fresh indexes i and j;
- auxiliary inputs of non coordinator roles are annotated with the fresh index i;
- auxiliary outputs in non coordinator roles and the corresponding input in the coordinator are both annotated with the fresh index j;
- the first output auxiliary communications of the coordinator are indexed with $(i, true)$, where i is the fresh index corresponding to the target role;
- the last output auxiliary communications of the coordinator are indexed with $(i, false)$, where i is the fresh index corresponding to the target role.

As for DIOCs, annotated DPOCs inherit the semantics of DPOCs, since indexes are just decorations, with no effect on the behaviour. There are however a few tricky points. In particular, we have to clarify how indexes are managed when new constructs are introduced. In rule [IN] the assignment inherits the index from the input primitive. In rule [WHILE-UNFOLD] the body is copied together with its indexes. In rule [LEAD-UP], when applying the update $\mathcal{I}$, we annotate $\mathcal{I}$ with

indexes never used elsewhere and distinct, and then generate the indexes for its projection as described above. Also, we assign to the auxiliary communications introduced by rules [Lead-Up] and [Lead-NoUp] indexes never used elsewhere. Auxiliary communications introduced by rule [Up] instead have to use the index of the corresponding communication introduced by rule [Lead-Up] (the index can be passed by extending the communication label). We can now define DPOC events. As for DIOC events, DPOC events correspond to transitions of the DPOC.

Definition 15 (DPOC events). *An annotated* DPOC *network $\mathcal{N}$ contains the following events:*

Communication events: *a sending event $\xi : \overline{o^?}@r_2$ in role r_1 for each output $n : o^? : e$ $\mathtt{to}$ r_2 with global index ξ in role r_1; and a receiving event $\xi : o^?@r_1$ in role r_2 for each input $n : o^? : x$ $\mathtt{from}$ r_1 with global index ξ in role r_2; we also denote the sending event as f_ξ or $[f_\xi]_{r_1}$; and the receiving event as t_ξ or $[t_\xi]_{r_2}$. Sending and receiving events correspond to the transitions executing the communications.*

Assignment events: *an assignment event ε_ξ in role r for each assignment $n : x = e$ with global index ξ; the event corresponds to the transition executing the assignment.*

Scope events: *a scope initialisation event $\uparrow_\xi$ and a scope termination event $\downarrow_\xi$ for each $n :$ $\mathtt{scope}$ $@r$ $\{P\}$ $\mathtt{roles}$ $\{S\}$ or $n :$ $\mathtt{scope}$ $@r$ $\{P\}$ with global index ξ. Scope events with the same global index coincide, and thus the same event may belong to different roles; the scope initialisation event corresponds to the transition performing or not performing an update on the given scope for the role leading the update. The scope termination event is just an auxiliary event (related to the auxiliary interactions implementing the scope termination).*

If events: *a guard if-event ε_ξ in role r for each construct $n :$ $\mathtt{if}$ b $\{P\}$ $\mathtt{else}$ $\{P'\}$ with global index ξ; the guard-if event corresponds to the transition evaluating the guard of the if.*

While events: *a guard while-event ε_ξ in role r for each construct $n :$ $\mathtt{while}$ b $\{P\}$ with global index ξ; the guard-while event corresponds to the transition evaluating the guard of the while.*

Let $\mathsf{events}(\mathcal{N})$ denote the set of events of the network $\mathcal{N}$. A sending and a receiving event with either the same global index ξ or with global indexes differing only for replacing index i with (i, true) or (i, false) are called matching events. We denote with $\overline{\varepsilon}$ an event matching event ε. A communication event is either a sending event or a receiving event. A communication event is unmatched if there is no event matching it.

With a slight abuse of notation, we write $\mathsf{events}(P)$ to denote events originated by constructs in process P, assuming the network $\mathcal{N}$ to be understood.

We used the same notations for events of the DIOC and of the DPOC. Indeed, the two kinds of events are strongly related (cfr. Lemma 2).

We can now define the causality relation among DPOC events.

Definition 16 (DPOC **causality relation**). *Let us consider an annotated* DPOC *network $\mathcal{N}$. A causality relation $\leq_{\mathsf{DPOC}} \subseteq \mathsf{events}(\mathcal{N}) \times \mathsf{events}(\mathcal{N})$ is a partial order among events in $\mathcal{N}$. We define $\leq_{\mathsf{DPOC}}$ as the minimum partial order satisfying:*

Sequentiality: *Let $P'; P''$ be a subterm of* DPOC *network $\mathcal{N}$. If ε' is an event in P' and ε'' is an event in P'', both in the same role r, then $\varepsilon' \leq_{\mathsf{DPOC}} \varepsilon''$.*

Scope-coordinator: *Let $n : \mathtt{scope} \; @r \; \{P\} \; \mathtt{roles} \; \{S\}$ be a subterm of* DPOC *$\mathcal{N}$ in role r with global index ξ. If ε' is an event in P then $\uparrow_\xi \leq_{\mathsf{DPOC}} \varepsilon' \leq_{\mathsf{DPOC}} \downarrow_\xi$.*

Scope-simple: *Let $n : \mathtt{scope} \; @r \; \{P\}$ be a subterm of* DPOC *$\mathcal{N}$ in role r' with global index ξ. If ε' is an event in P then $\uparrow_\xi \leq_{\mathsf{DPOC}} \varepsilon' \leq_{\mathsf{DPOC}} \downarrow_\xi$.*

Synchronisation: *For each pair of events ε and ε', $\varepsilon \leq \varepsilon'$ implies $\overline{\varepsilon} \leq_{\mathsf{DPOC}} \varepsilon'$.*

If: *Let $n : \mathtt{if} \; b \; \{P\} \; \mathtt{else} \; \{P'\}$ be a subterm of* DPOC *network $\mathcal{N}$ with global index ξ, let ε_ξ be the guard if-event in role r, then for every event ε in P and for every event ε' in P' we have $\varepsilon_\xi \leq_{\mathsf{DPOC}} \varepsilon$ and $\varepsilon_\xi \leq_{\mathsf{DPOC}} \varepsilon'$.*

While: *Let $n : \mathtt{while} \; b \; \{P\}$ be a subterm of* DPOC *network $\mathcal{N}$ with global index ξ, let ε_ξ be the guard while-event in role r, then for every event ε in P we have $\varepsilon_\xi \leq_{\mathsf{DPOC}} \varepsilon$.*

Lemma 2. *Given a* DIOC *process $\mathcal{I}$, for each state Σ the* DPOC *network $\mathsf{proj}(\mathcal{I}, \Sigma)$ is such that:*

1. $\mathsf{events}(\mathcal{I}) \subseteq \mathsf{events}(\mathsf{proj}(\mathcal{I}, \Sigma))$*;*
2. $\forall \varepsilon_1, \varepsilon_2 \in \mathsf{events}(\mathcal{I}). \varepsilon_1 \leq_{\mathsf{DIOC}} \varepsilon_2 \Rightarrow \varepsilon_1 \leq_{\mathsf{DPOC}} \varepsilon_2 \vee \varepsilon_1 \leq_{\mathsf{DPOC}} \overline{\varepsilon_2}$

Proof. 1. By definition of projection.

2. Let $\varepsilon_1 \leq_{\mathsf{DIOC}} \varepsilon_2$. We have a case analysis on the condition used to derive the dependency.

Sequentiality: Consider $\mathcal{I} = \mathcal{I}'; \mathcal{I}''$. If events are in the same role the implication follows from the sequentiality of the $\leq_{\mathsf{DPOC}}$.

Let us show that there exists an event ε'' in an initial interaction of $\mathcal{I}''$ such that either $\varepsilon'' \leq_{\mathsf{DPOC}} \varepsilon_2$ or $\varepsilon'' \leq_{\mathsf{DPOC}} \overline{\varepsilon_2}$. The proof is by induction on the structure of $\mathcal{I}''$. The only difficult case is sequential composition. Assume $\mathcal{I}'' = \mathcal{I}_1; \mathcal{I}_2$. If $\varepsilon_2 \in \mathsf{events}(\mathcal{I}_1)$ the thesis follows from inductive hypothesis. If $\varepsilon_2 \in \mathsf{events}(\mathcal{I}_2)$ then by induction there exists an event ε_3 in an initial interaction of $\mathcal{I}_2$ such that $\varepsilon_3 \leq_{\mathsf{DPOC}} \varepsilon_2$ or $\varepsilon_3 \leq_{\mathsf{DPOC}} \overline{\varepsilon_2}$. By synchronisation (Definition 16) we have that $\overline{\varepsilon_3} \leq_{\mathsf{DPOC}} \varepsilon_2$ or $\overline{\varepsilon_3} \leq_{\mathsf{DPOC}} \overline{\varepsilon_2}$. By connectedness for sequence we have that ε_3 or $\overline{\varepsilon_3}$ are in the same role of an event ε_4 in $\mathcal{I}'$. By sequentiality (Definition 16) we have that $\varepsilon_4 \leq_{\mathsf{DPOC}} \varepsilon_3$ or $\varepsilon_4 \leq_{\mathsf{DPOC}} \overline{\varepsilon_3}$. By synchronisation we have that $\overline{\varepsilon_4} \leq_{\mathsf{DPOC}} \varepsilon_3$ or $\overline{\varepsilon_4} \leq_{\mathsf{DPOC}} \overline{\varepsilon_3}$. The thesis follows from the inductive hypothesis on ε_4 and by transitivity of $\leq_{\mathsf{DPOC}}$.

Let us also show that there exists a final event $\varepsilon''' \in \mathsf{events}(\mathcal{I}')$ such that $\varepsilon_1 \leq_{\mathsf{DPOC}} \varepsilon'''$ or $\varepsilon_1 \leq_{\mathsf{DPOC}} \overline{\varepsilon'''}$. The proof is by induction on the structure of $\mathcal{I}'$. The only difficult case is sequential composition. Assume $\mathcal{I}' = \mathcal{I}_1; \mathcal{I}_2$. If $\varepsilon_1 \in \mathsf{events}(\mathcal{I}_2)$ the thesis follows from inductive hypothesis. If $\varepsilon_1 \in \mathsf{events}(\mathcal{I}_1)$ then the proof is similar to the one above,

finding a final event in $\mathcal{I}_1$ and applying sequentiality, synchronisation, and transitivity.

The thesis follows from the two results above again by sequentiality, synchronisation, and transitivity.

Scope: it means that either (1) $\varepsilon_1 = \uparrow_n$ and ε_2 is an event in the scope or (2) $\varepsilon_1 = \uparrow_n$ and $\varepsilon_2 = \downarrow_n$, or (3) ε_1 is an event in the scope and $\varepsilon_2 = \downarrow_n$. We consider the first case since the third one is analogous and the second one follows by transitivity. If ε_2 is in the coordinator then the thesis follows easily. Otherwise it follows thanks to the auxiliary synchronisations with a reasoning similar to the one for sequentiality.

Synchronisation: it means that ε_1 is a sending event and ε_2 is the corresponding receiving event, namely $\varepsilon_1 = \overline{\varepsilon_2}$. Thus, since $\varepsilon_2 \leq_{\mathsf{DPOC}} \varepsilon_2$ then $\overline{\varepsilon_2} \leq_{\mathsf{DPOC}} \varepsilon_2$.

If: it means that ε_1 is the evaluation of the guard and ε_2 is in one of the two branches. Thus, if ε_2 is in the coordinator then the thesis follows easily. Otherwise it follows thanks to the auxiliary synchronisations with a reasoning similar to the one for sequentiality.

While: it means that ε_1 is the evaluation of the guard and ε_2 is in the body of the while. Thus, if ε_2 is in the coordinator then the thesis follows easily. Otherwise it follows thanks to the auxiliary synchronisations with a reasoning similar to the one for sequentiality.

We can now define a notion of conflict between (DIOC and DPOC) events, relating events which are in different branches of the same conditional.

Definition 17 (Conflicting events). *Given a DIOC process $\mathcal{I}$ we say that two events $\varepsilon, \varepsilon' \in \mathsf{events}(\mathcal{I})$ are conflicting if they belong to different branches of the same if construct, i.e. there exists a subprocess* `if` b $\{\mathcal{I}'\}$ `else` $\{\mathcal{I}''\}$ *of $\mathcal{I}$ such that $\varepsilon \in \mathsf{events}(\mathcal{I}') \wedge \varepsilon' \in \mathsf{events}(\mathcal{I}'')$ or $\varepsilon' \in \mathsf{events}(\mathcal{I}') \wedge \varepsilon \in \mathsf{events}(\mathcal{I}'')$.*

Similarly, given a DPOC network $\mathcal{N}$, we say that two events $\varepsilon, \varepsilon' \in \mathsf{events}(\mathcal{N})$ are conflicting if they belong to different branches of the same if construct, i.e. there exists a subprocess `if` b $\{P\}$ `else` $\{P'\}$ *of $\mathcal{N}$ such that $\varepsilon \in \mathsf{events}(P) \wedge \varepsilon' \in \mathsf{events}(P')$ or $\varepsilon' \in \mathsf{events}(P) \wedge \varepsilon \in \mathsf{events}(P')$.*

DPOCs resulting from the projection of well-annotated connected DIOCs enjoy useful properties.

Definition 18 (Well-annotated DPOC). *An annotated DPOC network $\mathcal{N}$ is well-annotated for its causality relation $\leq_{\mathsf{DPOC}}$ if the following conditions hold:*

C1 *for each global index ξ there are at most two communication events with global index ξ and, in this case, they are matching events;*

C2 *only events which are minimal according to $\leq_{\mathsf{DPOC}}$ may correspond to enabled transitions;*

C3 *for each pair of non-conflicting sending events $[f_\xi]_r$ and $[f_{\xi'}]_r$ on the same operation $o^?$ with the same target s such that $\xi \neq \xi'$ we have $[f_\xi]_r \leq_{\mathsf{DPOC}} [f_{\xi'}]_r$ or $[f_{\xi'}]_r \leq_{\mathsf{DPOC}} [f_\xi]_r$;*

C4 *for each pair of non-conflicting receiving events $[t_\xi]_s$ and $[t_{\xi'}]_s$ on the same operation $o^?$ with the same sender r such that $\xi \neq \xi'$ we have $[t_\xi]_s \leq [t_{\xi'}]_s$ or $[t_{\xi'}]_s \leq [t_\xi]_s$;*

C5 *if ε is an event inside a scope with global index ξ then its matching event $\overline{\varepsilon}$ (if it exists) is inside a scope with the same global index.*

C6 *if two events have the same index but different global indexes then one of them is inside a while with global index ξ_1, let us call it ε_1, and the other, ε_2, is not. Furthermore, $\varepsilon_2 \leq_{\mathsf{DPOC}} \varepsilon_{\xi_1}$ where ε_{ξ_1} is the guarding while-event of the while with global index ξ_1.*

Update, conditional choice, and iteration at the DIOC level happen in one step, while they correspond to many steps of the projected DPOC. Also, scope execution introduces auxiliary communications which have no correspondence in the DIOC. Thus, we define the function upd that bridges this gap. More precisely, function upd is obtained as the composition of two functions, a function prop that completes the execution of DIOC actions which have already started, and a function sim that eliminates all the auxiliary closing communications.

Definition 19 (upd function). *Let $\mathcal{N}$ be an annotated DPOC (we drop annotations if not relevant). The upd function is defined as the composition of a function prop and a function sim. Thus, $\mathsf{upd}(\mathcal{N}) = \mathsf{sim}(\mathsf{prop}(\mathcal{N}))$. Network $\mathsf{prop}(\mathcal{N})$ is obtained from $\mathcal{N}$ by repeating the following operations while possible:*

1. *for each $o_n^* : \mathit{true}$ `to` r' enabled, replace every $o_n^* : x_n$ `from` r; `while` x_n $\{P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` $r\}$ not inside another while construct, with $P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` r; `while` x_n $\{P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` $r\}$; and replace $o_n^* : \mathit{true}$ `to` r' with $\mathbf{1}$.*

2. *for each $o_n^* : \mathit{false}$ `to` r' enabled, replace every $o_n^* : x_n$ `from` r; `while` x_n $\{P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` $r\}$ not inside another while construct, with $\mathbf{1}$; and replace $o_n^* : \mathit{false}$ `to` r' with $\mathbf{1}$.*

3. *for each `while` x_n $\{P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` $r\}$ enabled not inside another while construct, such that x_n evaluates to $\mathbf{\mathit{true}}$ in the local state, replace it with $P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` r; `while` x_n $\{P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` $r\}$.*

4. *for each `while` x_n $\{P; o_n^* : $ `ok` `to` $r; o_n^* : x_n$ `from` $r\}$ enabled not inside another while construct, such that x_n evaluates to $\mathbf{\mathit{false}}$ in the local state, replace it with $\mathbf{1}$.*

5. *for each $o_n^* : \mathit{true}$ `to` r' enabled, replace every $o_n^* : x_n$ `from` r; `if` x_n $\{P'\}$ `else` $\{P''\}$ not inside a while construct, with P'; and replace $o_n^* : \mathit{true}$ `to` r' with $\mathbf{1}$.*

6. *for each $o_n^* : \mathit{false}$ `to` r' enabled, replace every $o_n^* : x_n$ `from` r; `if` x_n $\{P'\}$ `else` $\{P''\}$ not inside a while construct, with P''; and replace $o_n^* : \mathit{false}$ `to` r' with $\mathbf{1}$.*

7. *for each `if` x_n $\{P'\}$ `else` $\{P''\}$ enabled such that x_n evaluates to $\mathbf{\mathit{true}}$ in the local state, replace it with P'.*

8. *for each `if` x_n $\{P'\}$ `else` $\{P''\}$ enabled such that x_n evaluates to $\mathbf{\mathit{false}}$ in the local state, replace it with P''.*

9. *for each $o_n^* : P$ `to` s enabled, replace every $n : $ `scope` `@`r $\{P'\}$ in role s not inside a while construct, with P, and replace $o_n^* : P$ `to` s with $\mathbf{1}$.*

*10. for each o_n^* : **no** **to** s enabled, replace every n : **scope** @r $\{P'\}$ in the role s not inside a while construct, with P' and replace o_n^* : P **to** s with $\mathbf{1}$.*

Network $\mathsf{sim}(\mathcal{N})$ *is obtained from* $\mathcal{N}$ *by repeating the following operations while possible:*

- *replace each o_n^* : **ok** **to** r, o_n^* : **ok** **to** r, o_n^* : _ **from** r or o_n^* : _ **from** r not inside a while construct with $\mathbf{1}$.*
- *replace each operation occurrence of the form $n \cdot o^?$ with $o^?$.*

Furthermore sim *may apply 0 or more times the following operation:*

- *replace a subterm $\mathbf{1}; P$ by P or a subterm $\mathbf{1} \mid P$ by P.*

The result below proves that in a well-annotated DPOC only transitions corresponding to events minimal w.r.t. the causality relation $\leq_{\mathsf{DPOC}}$ may be enabled.

Lemma 3. *If $\mathcal{N}$ is a DPOC, $\leq_{\mathsf{DPOC}}$ its causality relation and ε is an event corresponding to a transition enabled in $\mathcal{N}$ then ε is minimal w.r.t. $\leq_{\mathsf{DPOC}}$.*

Proof. The proof is by contradiction. Suppose ε is enabled but not minimal, i.e. there exists ε' such that $\varepsilon' \leq_{\mathsf{DPOC}} \varepsilon$. If there is more than one such ε' consider the one such that the length of the derivation of $\varepsilon' \leq_{\mathsf{DPOC}} \varepsilon$ is minimal. This derivation should have length one, and following Definition 16 it may result from one of the following cases:

- Sequentiality: $\varepsilon' \leq_{\mathsf{DPOC}} \varepsilon$ means that $\varepsilon' \in \mathsf{events}(P')$, $\varepsilon \in \mathsf{events}(P'')$, and $P'; P''$ is a subterm of $\mathcal{N}$. Because of the semantics of sequential composition ε cannot be enabled.
- Scope: let n : **scope** @r $\{P\}$ **roles** $\{S\}$ or n : **scope** @r $\{P\}$ be a subprocess of $\mathcal{N}$ with global index ξ. We have the following cases:
  - $\varepsilon' = \uparrow_\xi$ and $\varepsilon \in \mathsf{events}(P)$, and this implies that ε cannot be enabled since if ε' is enabled then the rules [UP] or [NOUP] for the evolution of the scope have not been applied yet;
  - $\varepsilon' = \uparrow_\xi$ and $\varepsilon = \downarrow_\xi$: this is trivial, since $\downarrow_\xi$ is an auxiliary event and no transition corresponds to it;
  - $\varepsilon' \in \mathsf{events}(P)$ and $\varepsilon = \downarrow_\xi$, but this is impossible since if ε' is enabled there is no event ε because the events $\uparrow_\xi$ and $\downarrow_\xi$ disappear as soon as the rule [LEAD-UP] or [LEAD-NOUP] is performed.
- If: $\varepsilon \leq_{\mathsf{DPOC}} \varepsilon'$ means that ε is the evaluation of the guard of the subterm n : **if** x_n $\{P'\}$ **else** $\{P''\}$ and $\varepsilon' \in \mathsf{events}(P') \cup \mathsf{events}(P'')$. Event ε' cannot be enabled because of the semantics of if.
- While: $\varepsilon \leq_{\mathsf{DPOC}} \varepsilon'$ means that ε is the evaluation of the guard of the subterm n : **while** x_n $\{P\}$ and $\varepsilon' \in \mathsf{events}(P)$. Event ε' cannot be enabled because of the semantics of while.

The following result shows that if an interaction is performed then the two executed events are matching events.

33

Lemma 4. *If $\mathcal{N}$ is a well-annotated* DPOC *and* $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{o^?:r_1(v) \to r_2(x)} \langle \mathbf{I}, \mathcal{N}' \rangle$ *then the two executed events are matching events.*

Proof. By definition of DPOC semantics we have that the transition $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{o^?:r_1(v) \to r_2(x)} \langle \mathbf{I}, \mathcal{N}' \rangle$ can be generated only by the [SYNCH] rule. Then, we have that the two events are on the same operation and that r_2 is the target of the first event. Assume that they are not matching events. Then for the definition of well-annotated DPOC, they are either conflicting or in the causality relation. In the first case, none of them can be enabled by Definition 16 since they are inside an *if* construct. In the second case thanks to Lemma 3, at least one of them cannot be enabled since it is not minimal. This is a contradiction, thus they are matching events. ∎

We now prove that all the DPOCs obtained as projection of well-annotated connected DIOCs are well-annotated.

Lemma 5. *Let $\mathcal{I}$ be a well-annotated connected* DIOC *process, and Σ a state. Then its projection $\mathcal{N} = \mathsf{proj}(\mathcal{I}, \Sigma)$ is a well-annotated* DPOC *network w.r.t.* $\leq_{\mathsf{DPOC}}$.

Proof. We have to prove that $\mathsf{proj}(\mathcal{I}, \Sigma)$ satisfies the conditions of Definition 18 of well-annotated DPOC:

C1 For each global index ξ there are at most two communication events with global index ξ and, in this case, they are matching events. The condition follows by the definition of the projection function, observing that in well-annotated DIOCs, each construct has its own index, and different indexes are mapped to different global indexes. Note that the two auxiliary input communications in the projection of a while construct on a non coordinating role have the same index but different global indexes.

C2 Only events which are minimal according to $\leq_{\mathsf{DPOC}}$ may correspond to enabled transitions. This condition follows from Lemma 3.

C3 For each pair of non-conflicting sending events $[f_\xi]_r$ and $[f_{\xi'}]_r$ on the same operation $o^?$ and with the same target such that $\xi \neq \xi'$ we have $[f_\xi]_r \leq_{\mathsf{DPOC}} [f_{\xi'}]_r$ or $[f_{\xi'}]_r \leq_{\mathsf{DPOC}} [f_\xi]_r$. Note that the two events are in the same role, thus w.l.o.g. we can assume that there exist two processes P, P' such that $[f_\xi]_r \in \mathsf{events}(P)$ and $[f_{\xi'}]_r \in \mathsf{events}(P')$ and that one among $P; P'$, $P|P'$, and `if` b $\{P\}$ `else` $\{P'\}$ is a subprocess of $\mathcal{N}$.

Since $\mathcal{I}$ is connected for parallel, by Definition 1 and by definition of the projection function the second case can never happen. Similarly, since the events are non-conflicting by Definition 17 the third case can never happen. If $P; P'$ is a subprocess of $\mathcal{N}$ then by sequentiality (Definition 16) we have the thesis.

C4 Similar to the previous case.

C5 By definition of the projection function.

C6 By definition of well-annotated DIOC and of projection the only case where there are two events with the same index and different global indexes is for the auxiliary communications in the projection of the while construct, where the conditions hold by construction.

The next lemma shows that for every set of updates $\mathbf{I}$ the DPOC $\mathcal{N}$ and $\mathsf{upd}(\mathcal{N})$ have the same set of weak traces.

Lemma 6. *Let $\mathcal{N}$ be a DPOC. The following properties hold:*

1. *if $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N}) \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \rangle$ with $\eta \in \{o^? : r_1(v) \to r_2(x), \sqrt{}, \mathcal{I}, \mathsf{no\text{-}up}, \tau\}$ then there exist $\mathcal{N}''$ s.t. $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta_1} \ldots \xrightarrow{\eta_k} \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}'' \rangle$ where $\eta_i \in \{o^* : r_1(v) \to r_2(x), \tau\}$ and $\mathsf{upd}(\mathcal{N}'') = \mathsf{upd}(\mathcal{N}')$.*
2. *if $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \rangle$ for $\eta \in \{o^? : r_1(v) \to r_2(x), \sqrt{}, \mathcal{I}, \mathsf{no\text{-}up}, \tau\}$, then one of the following holds: (A) $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N}) \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}'' \rangle$ such that $\mathsf{upd}(\mathcal{N}') = \mathsf{upd}(\mathcal{N}'')$, or (B) $\mathsf{upd}(\mathcal{N}) = \mathsf{upd}(\mathcal{N}')$ and $\eta \in \{o^* : r_1(v) \to r_2(x), \tau\}$;*

Proof.

1. The upd function corresponds to perform weak transitions, namely transitions with labels in $\{o^* : r_1(v) \to r_2(x), \tau\}$. $\mathcal{N}$ may perform the enabled weak transitions that correspond to the application of upd reducing to $\mathcal{N}'''$. Then, η is enabled also in $\mathcal{N}'''$ and we have $\langle \mathbf{I}, \mathcal{N}''' \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}'' \rangle$. At this point we have that $\mathcal{N}''$ and $\mathcal{N}'$ may differ only for communication primitives corresponding to weak transitions, removed by upd.
2. Either the transition with label η corresponds to one of the transitions executed by function upd or not. In the first case statement (B) holds trivially. Otherwise transition labeled by η is still enabled in $\mathsf{upd}(\mathcal{N})$ and the thesis follows.

We now prove a few properties of transitions with label $\sqrt{}$.

Lemma 7. *If $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ has a transition with label $\sqrt{}$ then, for each role $s \in \mathsf{roles}(\mathcal{I})$, $(\pi(\mathcal{I}, s), \Sigma_s)_s$ has a transition with label $\sqrt{}$ and vice versa.*

Proof. By structural induction on $\mathcal{I}$.

The next lemma shows that if two matching events are enabled in the projection of a DIOC, then the corresponding interaction is enabled in the DIOC.

Lemma 8. *Let $\mathcal{I}$ be a DIOC obtained from a well-annotated connected DIOC via 0 or more transitions and $n : o^? : r_1(e) \to r_2(x)$ be an interaction in $\mathcal{I}$. If $n : o^? : e$ to r_1 and $n : o^? : x$ from r_2 are matching events and are both enabled in $\mathsf{proj}(\mathcal{I}, \Sigma)$ then $n : o^? : r_1(e) \to r_2(x)$ is enabled.*

Proof. Note that $\mathcal{I}$ is well-annotated and connected for parallel.

If $\mathcal{I}$ is also connected for sequence, then the proof is by structural induction on $\mathcal{I}$. The cases for **1**, **0**, and scopes, if, and while constructs are trivial. For parallel composition just consider that since the two events have the same global index then they are from the same component, and the thesis follows by inductive hypothesis. Let us consider sequential composition. Suppose $\mathcal{I} = \mathcal{I}'; \mathcal{I}''$. If $n : o^? : r_1(e) \to r_2(x) \in \mathcal{I}'$ then the thesis follows by inductive hypothesis. Otherwise, by inductive hypothesis $n : o^? : r_1(e) \to r_2(x)$ is enabled in $\mathcal{I}''$. Thus, $r_1 \to r_2 \in \mathsf{transl}(\mathcal{I}'')$. From connectedness for sequence $\forall s_1 \to s_2 \in \mathsf{transF}(\mathcal{I}')$ then $\{r_1, r_2\} \cap \{s_1, s_2\} \neq \emptyset$. This is not possible since otherwise at least one of the events $n : o^? : e \; \mathtt{to} \; r_1$ and $n : o^? : x \; \mathtt{from} \; r_2$ would not be enabled. Thus, the only possibility is $\mathsf{transF}(\mathcal{I}') = \emptyset$. This implies that $\mathcal{I}'$ has a transition with label $\sqrt{}$. Thus, $n : o^? : r_1(e) \to r_2(x)$ is enabled in $\mathcal{I}$.

If $\mathcal{I}$ is not connected for sequence, then in the projected DPOC some more transitions may be enabled, but no required transitions may be disabled, thus the thesis follows.

Definition 20 (Weak System Bisimilarity). *A weak system bisimulation is a relation R between* DIOC *systems and* DPOC *systems such that if $(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}', \mathcal{N} \rangle) \in R$ then:*

- *if $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle$ then $\langle \mathbf{I}', \mathcal{N} \rangle \xrightarrow{\eta_1} \ldots \xrightarrow{\eta_k} \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}''' \rangle$ with $\forall i \in [1..k], \eta_i \in \{o^* : r_1(v) \to r_2(x), \tau\}$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle) \in R$ and $\eta = \mu$ or $\eta = n \cdot o^? : r_1(v) \to r_2(x)$ and $\mu = o^? : r_1(v) \to r_2(x);$*

- *if $\langle \mathbf{I}', \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}''' \rangle$ with $\eta \in \{o^? : r_1(v) \to r_2(x); \sqrt{}; \mathcal{I}; \mathsf{no\text{-}up}; \mathbf{I}''', \tau\}$ then one of the following two holds:*

  - *$\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}, \mathcal{I}'' \rangle$, with $\eta = \mu$ or $\eta = n \cdot o^? : r_1(v) \to r_2(x)$ and $\mu = o^? : r_1(v) \to r_2(x)$ and it holds that $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle) \in R;$*
  - *$\eta \in \{o^* : r_1(v) \to r_2(x), o^* : r_1(X) \to r_2(\_), \tau\}$ and it holds that $(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}''', \mathcal{N}'' \rangle) \in R$*

Weak system bisimilarity $\sim$ is the largest weak system bisimulation.

The following result states that weak system bisimilarity implies weak trace equivalence.

Lemma 9. *Let $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ be a* DIOC *system and $\langle \mathbf{I}', \mathcal{N} \rangle$ a* DPOC *system. If $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \sim \langle \mathbf{I}', \mathcal{N} \rangle$ then the* DIOC *system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ and the* DPOC *system $\langle \mathbf{I}', \mathcal{N} \rangle$ are weak trace equivalent.*

Proof. The proof is by coinduction. Take a DIOC trace $\mu_1, \mu_2, \ldots$ of the DIOC system. From bisimilarity, the DPOC system has a transition with label η_1 matching μ_1. After the transition, the DIOC system and the DPOC system are again

bisimilar. Thus the DPOC system has a trace $\eta_2, \ldots$ matching $\mu_2, \ldots$. By composition the DPOC system has a trace $\eta_1, \eta_2, \ldots$ as desired. The opposite direction is analogous.

We can now prove our main theorem, that states that given a connected well-annotated DIOC process $\mathcal{I}$ and a state Σ the DPOC network obtained as its projection has the same behaviours of $\mathcal{I}$.

Theorem 2. *For each initial, connected DIOC process $\mathcal{I}$, each state Σ, and each set of updates $\mathbf{I}$, the DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ and the DPOC system $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ are weak trace equivalent.*

Proof. We prove that the relation R below is a weak system bisimulation.

$$
R = \left\{ \left(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}, \mathcal{N} \rangle \right) \middle|
\begin{array}{l}
\mathsf{upd}(\mathcal{N}) = \mathsf{proj}(\mathcal{I}, \Sigma), \\
\mathsf{events}(\mathcal{I}) \subseteq \mathsf{events}(\mathsf{prop}(\mathcal{N})), \\
\forall \varepsilon_1, \varepsilon_2 \in \mathsf{events}(\mathcal{I}). \\
\varepsilon_1 \leq_{\mathsf{DIOC}} \varepsilon_2 \Rightarrow \\
\varepsilon_1 \leq_{\mathsf{DPOC}} \varepsilon_2 \vee \varepsilon_1 \leq_{\mathsf{DPOC}} \overline{\varepsilon_2}
\end{array}
\right\}
$$

where $\mathcal{I}$ is obtained from a well-annotated connected DIOC via 0 or more transitions and $\mathsf{upd}(\mathcal{N})$ is a well-annotated DPOC.

To ensure that proving that the relation above is a bisimulation implies our thesis, let us show that the pair $\left(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \right)$ from the theorem statement belongs to R. Note that here $\mathcal{I}$ is well-annotated and connected, and for each such $\mathcal{I}$ we have $\mathsf{upd}(\mathsf{proj}(\mathcal{I}, \Sigma)) = \mathsf{proj}(\mathcal{I}, \Sigma)$. From Lemma 5 $\mathsf{proj}(\mathcal{I}, \Sigma)$ is well annotated, thus $\mathsf{upd}(\mathsf{proj}(\mathcal{I}, \Sigma))$ is well annotated.

Observe that prop is the identity on $\mathsf{proj}(\mathcal{I}, \Sigma)$, thus from Lemma 2 we have that the conditions $\mathsf{events}(\mathcal{I}) \subseteq \mathsf{events}(\mathsf{prop}(\mathcal{N}))$ and $\forall \varepsilon_1, \varepsilon_2 \in \mathsf{events}(\mathcal{I}). \varepsilon_1 \leq_{\mathsf{DIOC}} \varepsilon_2 \Rightarrow \varepsilon_1 \leq_{\mathsf{DPOC}} \varepsilon_2 \vee \varepsilon_1 \leq_{\mathsf{DPOC}} \overline{\varepsilon_2}$ are satisfied. We now prove that R is a weak system bisimulation. From Lemma 9, this implies weak trace equivalence.

To prove that R is a weak system bisimulation it is enough to prove that for each $\left(\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle, \langle \mathbf{I}, \mathcal{N} \rangle \right)$ where $\mathcal{N} = \mathsf{proj}(\mathcal{I}, \Sigma)$ we have:

- if $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle$ then
  $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}''' \rangle$
  with $\left(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle \right) \in R$ and
  $\eta = \mu$ or $\eta = n \cdot o^? : r_1(v) \to r_2(x)$ and $\mu = o^? : r_1(v) \to r_2(x)$;
- if $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}''' \rangle$ with
  $\eta \in \{ o^? : r_1(v) \to r_2(x); \sqrt{}; \mathcal{I}; \mathsf{no\text{-}up};\, , \mathbf{I}'''; \tau \}$ then
  $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}, \mathcal{I}'' \rangle$ and
  $\left(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}''', \mathcal{N}''' \rangle \right) \in R$ and $\eta = \mu$ or $\eta = n \cdot o^? : r_1(v) \to r_2(x)$ and $\mu = o^? : r_1(v) \to r_2(x)$.

In fact, consider $\mathcal{N}$ with $\mathsf{upd}(\mathcal{N}) = \mathsf{proj}(\mathcal{I}, \Sigma)$. The case for labels $\Sigma, \mathbf{I}$ is trivial. If $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle$, then by hypothesis $\mathsf{upd}(\mathcal{N}) \xrightarrow{\eta} \mathcal{N}'''$. The thesis follows from Lemma 6 (case one). If instead $\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}''', \mathcal{N}''' \rangle$ with $\eta \in \{ o^? :$

37

$r_1(v) \to r_2(x), \surd, \mathcal{I}, \mathsf{no\text{-}up}, \tau\}$ then thanks to Lemma 6 we have one of the following: (A) $\mathsf{upd}(\mathcal{N}) \xrightarrow{\eta} \mathcal{N}''$ such that $\mathsf{upd}(\mathcal{N}''') = \mathsf{upd}(\mathcal{N}'')$, or (B) $\mathsf{upd}(\mathcal{N}) = \mathsf{upd}(\mathcal{N}''')$ and $\eta \in \{o^* : r_1(v) \to r_2(x), o^* : r_1(X) \to r_2(\_), \tau\}$. In case (A) we have $\langle \mathbf{I}, \mathsf{upd}(\mathcal{N}) \rangle \xrightarrow{\eta} \langle \mathbf{I}'', \mathcal{N}'' \rangle$. Then we have $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\mu} \langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle$ and $(\langle \Sigma'', \mathbf{I}'', \mathcal{I}'' \rangle, \langle \mathbf{I}'', \mathcal{N}'' \rangle) \in R$. The thesis follows since $\mathsf{upd}(\mathcal{N}''') = \mathsf{upd}(\mathcal{N}'')$. In case (B) the step is matched by the DIOC by staying idle, following the second option in the definition of weak system bisimilarity.

Thus, we have to prove the two conditions above. The proof is by structural induction on the DIOC $\mathcal{I}$. All the subterms of a well-annotated connected DIOC are well-annotated and connected, thus the induction can be performed. We consider both challenges from the DIOC ($\to$) and from the DPOC ($\leftarrow$). The case for label $\surd$ follows from Lemma 7. The case for labels $\Sigma, \mathbf{I}$ is trivial. Let us consider the other labels, namely $o^? : r_1(v) \to r_2(x), \mathcal{I}, \mathsf{no\text{-}up}$, and τ.

Note that no transition (at the DIOC or at the DPOC level) with one of these labels can change the set of updates $\mathbf{I}$. Thus, in the following, we will not write it. Essentially, we will use DIOC processes and DPOC networks instead of DIOC systems and DPOC systems respectively. Note that DPOC networks also include the state, while this is not the case for DIOC processes. For DIOC processes, we assume to associate to them the state Σ, and comment on its changes whenever needed.

Case 1, 0: trivial.

Case $n : x@r = e$**:** the assignment changes the global state in the DIOC, and the local state of role r in the DPOC in a corresponding way.

Case $n : o^? : r_1(e) \to r_2(x)$**:** trivial unless the interaction has been created by an update step. In this last case, note that the mismatch on the name of the operation, namely between $n \cdot o^?$ in the DPOC and $o^?$ in the DIOC, is solved thanks to the definition of weak system bisimilarity.

Case $\mathcal{I}; \mathcal{I}'$**:** from the definition of the projection function we have that
$$\mathcal{N} = \|_{r \in \mathsf{roles}(\mathcal{I}; \mathcal{I}')} (\pi(\mathcal{I}, r); \pi(\mathcal{I}', r), \Sigma_r)_r.$$

$\to$ Assume that $\mathcal{I}; \mathcal{I}' \xrightarrow{\mu} \mathcal{I}''$ with $\mu \in \{o^? : r_1(v) \to r_2(x); \mathcal{I}; \mathsf{no\text{-}up}, \tau\}$. There are two possibilities: either $\mathcal{I} \xrightarrow{\mu} \mathcal{I}'''$ and $\mathcal{I}'' = \mathcal{I}'''; \mathcal{I}'$ or $\mathcal{I}$ has a transition with label $\surd$ and $\mathcal{I}' \xrightarrow{\mu} \mathcal{I}''$. In the first case by inductive hypothesis $\|_{r \in \mathsf{roles}(\mathcal{I})} (\pi(\mathcal{I}, r), \Sigma_r)_r \xrightarrow{\eta} \mathcal{N}'''$ with η corresponding to μ and $\mathsf{upd}(\mathcal{N}''') = \|_{r \in \mathsf{roles}(\mathcal{I})} (\pi(\mathcal{I}''', r), \Sigma_r')_r$. Thus $\|_{r \in \mathsf{roles}(\mathcal{I})} (\pi(\mathcal{I}, r); \pi(\mathcal{I}', r), \Sigma_r)_r \xrightarrow{\eta} \mathcal{N}$ and we have $\mathsf{upd}(\mathcal{N}) = \|_{r \in \mathsf{roles}(\mathcal{I})} (\pi(\mathcal{I}''', r); \pi(\mathcal{I}', r), \Sigma_r')_r$. If $\mathsf{roles}(\mathcal{I}') \subseteq \mathsf{roles}(\mathcal{I})$ then the thesis follows. Otherwise roles in $\mathsf{roles}(\mathcal{I}') \setminus \mathsf{roles}(\mathcal{I})$ are unchanged. Note however that the projection of $\mathcal{I}$ on these roles is a term composed only by $\mathbf{1}$s, which can be removed by function upd. If $\mathcal{I}$ has a transition with label $\surd$ and $\mathcal{I}' \xrightarrow{\mu} \mathcal{I}''$ then by inductive hypothesis $\mathsf{proj}(\mathcal{I}', \Sigma) \xrightarrow{\eta} \mathcal{N}''$ with η corresponding to μ and $\mathsf{upd}(\mathcal{N}'') = \mathsf{proj}(\mathcal{I}'', \Sigma')$. The thesis follows since, thanks to Lemma 7, $\mathsf{proj}(\mathcal{I}; \mathcal{I}', \Sigma) \xrightarrow{\eta} \mathcal{N}$ and $\mathsf{upd}(\mathcal{N}) = \mathsf{proj}(\mathcal{I}'', \Sigma')$.

Note that, in both the cases, conditions on events follow by inductive hypothesis.

$\leftarrow$ Assume that

$$\mathcal{N} = \|_{r \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} \ (\pi(\mathcal{I},r); \pi(\mathcal{I}',r), \Sigma_r)_r \xrightarrow{\eta} \|_{r \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} \ (P_r, \Sigma'_r)_r$$

with $\eta \in \{o^? : r_1(v) \to r_2(x), \mathcal{I}, \mathsf{no\text{-}up}, \tau\}$. We have a case analysis on η.
If $\eta = o^? : r_1(v) \to r_2(x)$ then $(\pi(\mathcal{I};\mathcal{I}', r_1), \Sigma_{r_1})_{r_1} \xrightarrow{\overline{o^?}\langle v \rangle @ r_2 : r_1} (P_{r_1}, \Sigma_{r_1})_{r_1}$
and also $(\pi(\mathcal{I};\mathcal{I}', r_2), \Sigma_{r_2})_{r_2} \xrightarrow{o^?(x \leftarrow v) @ r_1 : r_2} (P_{r_2}, \Sigma_{r_2})_{r_2}$. The two events should have the same global index thanks to Lemma 4. Thus, they are either both from $\mathcal{I}$ or both from $\mathcal{I}'$.
In the first case we have also

$$\|_{r \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} \ (\pi(\mathcal{I},r), \Sigma_r)_r \xrightarrow{o^? : r_1(v) \to r_2(x)} \|_{r \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} \ (P''_r, \Sigma'_r)_r$$

with $P_r = P''_r; \pi(\mathcal{I}', r)$. Thus, by inductive hypothesis, $\mathcal{I} \xrightarrow{o^? : r_1(v) \to r_2(x)} \mathcal{I}''$ and $\mathsf{upd}(\|_{r \in \mathsf{roles} \ \mathcal{I};\mathcal{I}'} \ (P''_r, \Sigma_r)_r)$ is the projection of $\mathcal{I}''$ with state Σ.
Hence, we have that $\mathcal{I}; \mathcal{I}' \xrightarrow{o^? : r_1(v) \to r_2(x)} \mathcal{I}''; \mathcal{I}'$.
In the second case, thanks to Lemma 8, we have that the interaction is enabled. Thus, $\mathcal{I}$ has a transition with label $\sqrt{}$ and $\mathcal{I}' \xrightarrow{o^? : r_1(v) \to r_2(x)} \mathcal{I}''$. Thanks to Lemma 7 then both $(\pi(\mathcal{I}, r_1), \Sigma_{r_1})_{r_1}$ and $(\pi(\mathcal{I}, r_2), \Sigma_{r_2})_{r_2}$ have a transition with label $\sqrt{}$. Thus, we have $(\pi(\mathcal{I}', r_1), \Sigma_{r_1})_{r_1} \xrightarrow{\overline{o^?}\langle v \rangle @ r_2 : r_1} (P_{r_1}, \Sigma_{r_1})_{r_1}, (\pi(\mathcal{I}', r_2), \Sigma_{r_2})_{r_2} \xrightarrow{o^?(x \leftarrow v) @ r_1 : r_2} (P_{r_2}, \Sigma_{r_2})_{r_2}$ and $\mathsf{proj}(\mathcal{I}', \Sigma) \xrightarrow{o^? : r_1(v) \to r_2(x)} \|_{r \in \mathsf{roles}(\mathcal{I}')} \ (P_r, \Sigma_r)_r$. The thesis follows by inductive hypothesis. If η uses an extended operation then the corresponding DIOC transition uses the corresponding basic operation.
For the other possibilities of η, only the process of one role changes. Thus, the thesis follows by induction.
Note that in all the above cases, conditions on events follow by inductive hypothesis.

Case $\mathcal{I}|\mathcal{I}'$: from the definition of the projection function we have
$\mathcal{N} = \|_{r \in \mathsf{roles}(\mathcal{I};\mathcal{I}')} \ (\pi(\mathcal{I},r) \mid \pi(\mathcal{I}',r), \Sigma_r)_r$.

$\rightarrow$ If $\mathcal{I}|\mathcal{I}'$ can perform a transition then one of its two components can perform the same transition and the thesis follows by inductive hypothesis. Additional roles not occurring in the term performing the transition are dealt with by function upd.

$\leftarrow$ We have a case analysis on η. If $\eta = o^? : r_1(v) \to r_2(x)$ then an input and an output on the same operation are enabled. Thanks to Lemma 4 they have the same global index. Thus they are from the same component and the thesis follows by inductive hypothesis. For the other possibilities of η, only the process of one role changes. The thesis follows by induction. In all the cases, roles not occurring in the term performing the transition are dealt with by function upd.

39

Case $n : \mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\}$**:** from the definition of projection

$$\mathcal{N} =\|_{s\in\mathsf{roles}(\mathcal{I})\cup\mathsf{roles}(\mathcal{I}')\smallsetminus\{r\}}\ (o_n^* : x_n\ \mathtt{from}\ r;$$
$$\mathtt{if}\ x_n\ \{\pi(\mathcal{I},s)\}\ \mathtt{else}\ \{\pi(\mathcal{I}',s)\}, \Sigma_s)_s\ \|$$
$$(\mathtt{if}\ b\ \{(\Pi_{r'\in\mathsf{roles}(\mathcal{I})\cup\mathsf{roles}(\mathcal{I}')\smallsetminus\{r\}}o_n^* : true\ \mathtt{to}\ r');\pi(\mathcal{I},r)\}$$
$$\mathtt{else}\ \{(\Pi_{r'\in\mathsf{roles}(\mathcal{I})\cup\mathsf{roles}(\mathcal{I}')\smallsetminus\{r\}}o_n^* : false\ \mathtt{to}\ r');\pi(\mathcal{I}',r)\}, \Sigma_r)_r$$

Let us consider the case when the guard is true (the other one is analogous).

$\rightarrow$ The only possible transition from the DIOC is $n : \mathtt{if}\ b@r\ \{\mathcal{I}\}\ \mathtt{else}\ \{\mathcal{I}'\} \xrightarrow{\tau}$ $\mathcal{I}$. The DPOC can match this transition by reducing to

$$\mathcal{N}' =\|_{s\in\mathsf{roles}(\mathcal{I})\cup\mathsf{roles}(\mathcal{I}')\smallsetminus\{r\}}\ (o_n^* : x_n\ \mathtt{from}\ r;$$
$$\mathtt{if}\ x_n\ \{\pi(\mathcal{I},s)\}\ \mathtt{else}\ \{\pi(\mathcal{I}',s)\}, \Sigma_s)_s\ \|$$
$$(\Pi_{r'\in\mathsf{roles}(\mathcal{I})\cup\mathsf{roles}(\mathcal{I}')\smallsetminus\{r\}}o_n^* : true\ \mathtt{to}\ r';\pi(\mathcal{I},r), \Sigma_r)_r$$

By applying function upd we get

$$\mathsf{upd}(\mathcal{N}')\ =\|_{s\in\mathsf{roles}(\mathcal{I})\cup\mathsf{roles}(\mathcal{I}')\smallsetminus\{r\}}\ (\pi(\mathcal{I},s), \Sigma_s)_s\ \|\ (\pi(\mathcal{I},r), \Sigma_r)_r$$

Concerning events, at the DIOC level events corresponding to the guard and to the non-chosen branch are removed. The same holds at the DPOC level, thus conditions on the remaining events are inherited. This concludes the proof.

$\leftarrow$ The only possible transition from the DPOC is the evaluation of the guard from the coordinator. This reduces $\mathcal{N}$ to $\mathcal{N}'$ above and the thesis follows from the same reasoning.

Case $n : \mathtt{while}\ b@r\ \{\mathcal{I}\}$**:** from the definition of projection

$$\mathcal{N} =\|_{s\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}\ (o^* : x_n\ \mathtt{from}\ r;$$
$$\mathtt{while}\ x_n\ \{\pi(\mathcal{I},s); o_n^* : \mathtt{ok}\ \mathtt{to}\ r; o_n^* : x_n\ \mathtt{from}\ r\}, \Sigma_s)_s\ \|$$
$$(\mathtt{while}\ b\ \{\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : true\ \mathtt{to}\ r';\pi(\mathcal{I},r);$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : \_\ \mathtt{from}\ r'\};$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : false\ \mathtt{to}\ r', \Sigma_r)_r$$

$\rightarrow$ Let us consider the case when the guard is true. The only possible transition from the DIOC is $n : \mathtt{while}\ b@r\ \{\mathcal{I}\} \xrightarrow{\tau} \mathcal{I}; n : \mathtt{while}\ b@r\ \{\mathcal{I}\}$. The DPOC can match this transition by reducing to

$$\mathcal{N}' =\|_{s\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}\ (o^* : x_n\ \mathtt{from}\ r;$$
$$\mathtt{while}\ x_n\ \{\pi(\mathcal{I},s); o_n^* : \mathtt{ok}\ \mathtt{to}\ r; o_n^* : x_n\ \mathtt{from}\ r\}, \Sigma_s)_s\ \|$$
$$(\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : true\ \mathtt{to}\ r';\pi(\mathcal{I},r);$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : \_\ \mathtt{from}\ r';$$
$$\mathtt{while}\ b\ \{\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : true\ \mathtt{to}\ r';\pi(\mathcal{I},r);$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : \_\ \mathtt{from}\ r'\};$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}o_n^* : false\ \mathtt{to}\ r', \Sigma_r)_r$$

By applying function upd we get

$$\mathsf{upd}(\mathcal{N}') = \|_{s\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} \big(\pi(\mathcal{I},s); o_n^* : x_n \ \mathtt{from}\ r;$$
$$\mathtt{while}\ x_n\ \{\pi(\mathcal{I},s); o_n^* : \mathtt{ok}\ \mathtt{to}\ r; o_n^* : x_n\ \mathtt{from}\ r\}, \Sigma_s\big)_s \ \|$$
$$\big(\pi(\mathcal{I},r); \mathtt{while}\ b\ \{\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} o_n^* : true\ \mathtt{to}\ r'; \pi(\mathcal{I},r);$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} o_n^* : \_\ \mathtt{from}\ r'\};$$
$$\Pi_{r'\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} o_n^* : false\ \mathtt{to}\ r', \Sigma_r\big)_r$$

which is exactly the projection of $\mathcal{I}; n : \mathtt{while}\ b@r\ \{\mathcal{I}\}$.

As far as events are concerned, in $\mathsf{prop}(\mathcal{N}')$ we have all the needed events since, in particular, we have already done the unfolding of the while in all the roles. Concerning the ordering, at the DIOC level, we have two kinds of causal dependencies: (1) events in the unfolded process precede the guard event; (2) the guard event precedes the events in the body. The first kind of causal dependency is matched at the DPOC level thanks to the auxiliary synchronisations that close the unfolded body (which are not removed by prop) using synchronisation and sequentiality. The second kind of causal dependency is matched thanks to the auxiliary synchronisations that start the following iteration using synchronisation, sequentiality and while.

The case when the guard evaluates to **false** is simpler.

← The only possible transition from the DPOC is the evaluation of the guard from the coordinator. This reduces $\mathcal{N}$ to $\mathcal{N}'$ above and the thesis follows from the same reasoning.

Case $n : \mathtt{scope}\ @r\ \{\mathcal{I}\}$: from the definition of the projection

$$\mathcal{N} = \|_{s\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} \big(n : \mathtt{scope}\ @r\ \{\pi(\mathcal{I},s)\}, \Sigma_s\big)_s \ \|$$
$$n : \mathtt{scope}\ @r\ \{\pi(\mathcal{I},r)\}\ \mathtt{roles}\ \{\mathsf{roles}(\mathcal{I})\}$$

→ The only possible transitions are obtained by applying rules [LEAD-UP] or [LEAD-NOUP] to the coordinator scope. Let us consider the first case.

$$\mathcal{N} = \|_{s\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} \big(n : \mathtt{scope}\ @r\ \{\pi(\mathcal{I},s)\}, \Sigma_s\big)_s \ \|$$
$$n : \mathtt{scope}\ @r\ \{\pi(\mathcal{I},r)\}\ \mathtt{roles}\ \{\mathsf{roles}(\mathcal{I})\}$$
$$\xrightarrow{\mathcal{I}'} \|_{s\in\mathsf{roles}(\mathcal{I})\cup\{r\}} (P_s, \Sigma_s)_s = \mathcal{N}'$$

For the coordinator we have:

$$P_r = \Pi_{r_i\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}}$$
$$o_n^* : \pi\big(\mathsf{freshIndex}(\mathcal{I}',n), r_i\big)\ \mathtt{to}\ r_i;$$
$$\pi\big(\mathsf{freshIndex}(\mathcal{I}',n), r\big);$$
$$\Pi_{r_i\in\mathsf{roles}(\mathcal{I})\smallsetminus\{r\}} o_n^* : \_\ \mathtt{from}\ r_i$$

For other roles $P_{r_i} = n : \texttt{scope } @r \ \{P\}$. By applying the upd function we get:

$$\mathsf{upd}(\mathcal{N}') = \pi(\mathsf{freshIndex}(\mathcal{I}', n), r) \ \|$$
$$\|_{r_i \in \mathsf{roles}(\mathcal{I}) \smallsetminus \{r\}} \ \pi(\mathsf{freshIndex}(\mathcal{I}', n), r_i)$$

This is exactly the projection of the DIOC obtained after applying the rule [UP]. The conditions on events are inherited. Observe that the closing event of the scope is replaced by events corresponding to the auxiliary interactions closing the scope. This allows us to preserve the causality dependencies also when the scope is inserted in a bigger context. The case of rule [LEAD-NOUP] is simpler.

$\leftarrow$ The only possible transition from the DPOC is the one of the coordinator of the scope checking whether to apply an update. This reduces $\mathcal{N}$ to $\mathcal{N}'$ above and the thesis follows from the same reasoning.

G Proof of Corollary 1

Before proving Corollary 1, we prove an auxiliary lemma.

Lemma 10. *For each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates $\mathbf{I}$, if $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle \xrightarrow{\sqrt{}} \langle \Sigma', \mathbf{I}', \mathcal{I}' \rangle$ then the only transitions of $\langle \Sigma', \mathbf{I}', \mathcal{I}' \rangle$ have label $\mathbf{I}''$ for some $\mathbf{I}''$.*

Proof. The proof is by case analysis on the rules which can derive a transition with label $\sqrt{}$. All the cases are easy.

Corollary 1. *For each initial, connected DIOC $\mathcal{I}$, state Σ, and set of updates $\mathbf{I}$ the DPOC system $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle$ is deadlock-free.*

Proof. A DIOC system $\langle \Sigma, \mathbf{I}, \mathcal{I} \rangle$ is deadlock-free if all its maximal finite internal traces have $\sqrt{}$ as label of the last transition. For each trace, the property can be proved by induction on its length, and for each length by structural induction on $\mathcal{I}$. The proof is based on the fact that $\mathcal{I}$ is initial. The induction considers a reinforced hypothesis, saying also that $\sqrt{}$ never occurs before the end of the internal trace and that all the steps, but the last one, lead to initial DIOCs. We have a case analysis on the top-level operator in $\mathcal{I}$. Note that in all the cases at least a transition is derivable.

Case 0: not allowed since we assumed an initial DIOC.
Case 1: trivial because by rule [END] and Lemma 10 its only internal trace is $\sqrt{}$.
Case $x@r = e$: the only applicable rule is [ASSIGN] that in one step leads to a **1** process. The thesis follows by inductive hypothesis on the length of the trace.

Case $o^? : r_1(e) \to r_2(x)$**:** the only applicable rule is [INTERACTION], which leads to an assignment. Then the thesis follows by inductive hypothesis on the length of the trace.

Case $\mathcal{I}; \mathcal{I}'$**:** the first transition can be derived either by rule [SEQUENCE] or [SEQ-END]. In the first case the thesis follows by induction on the length of the trace. In the second case the trace coincides with a trace of $\mathcal{I}'$, and the thesis follows by structural induction.

Case $\mathcal{I} | \mathcal{I}'$**:** the first transition can be derived either by rule [PARALLEL] or by rule [PAR-END]. In the first case the thesis follows by induction on the length of the trace. In the second case the thesis follows by Lemma 10, since the label is $\sqrt{}$.

Case if $b@r$ $\{\mathcal{I}\}$ **else** $\{\mathcal{I}'\}$**:** the first transition can be derived using either rule [IF-THEN] or rule [IF-ELSE]. In both the cases the thesis follows by induction on the length of the trace.

Case while $b@r$ $\{\mathcal{I}\}$**:** the first transition can be derived using either rule [WHILE-UNFOLD] or rule [WHILE-EXIT]. In both the cases the thesis follows by induction on the length of the trace.

Case scope $@r$ $\{\mathcal{I}\}$**:** the first rule applied is either [UP] or [NOUP]. In both the cases the thesis follows by induction on the length of the trace.

The weak internal traces of the DIOC coincide with the weak internal traces of the DPOC by Theorem 2, thus the finite weak internal traces end with $\sqrt{}$. The same holds for the finite (strong) internal traces, since label $\sqrt{}$ is preserved when moving between strong and weak traces, and no transition can be added after the $\sqrt{}$ thanks to Lemma 10.

H Proof of Corollary 2

Corollary 2 (Race-freedom). *For each initial, connected* DIOC $\mathcal{I}$*, state* Σ*, and set of updates* **I***, if* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} \langle \mathbf{I}', \mathcal{N} \rangle$*, then in* $\mathcal{N}$ *two outputs (resp. inputs) cannot interact with the same input (resp. output).*

Proof. The result follows from Lemma 4, which shows that a DPOC transition always executes two matching events, since for each input (resp. output) at most one matching output (resp. input) exists.

I Proof of Corollary 3

Corollary 3 (Orphan message-freedom). *For each initial, connected* DIOC $\mathcal{I}$*, state* Σ*, and set of updates* **I***, if* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \xrightarrow{\mu_1} \cdots \xrightarrow{\sqrt{}} \langle \mathbf{I}', \mathcal{N} \rangle$*, then* $\mathcal{N}$ *contains no outputs.*

Proof. The proof is by case analysis on the rules which can derive a transition with label $\sqrt{}$. All the cases are easy.