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 [18] 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.

* This work is partly supported by the MIUR FIRB project FACE (Formal Avenue for Chasing malwarE) RBFR13AJFT and by the Italian MIUR PRIN Project CINA Prot. 2010LHT4KM.

Here, we propose an approach based on *choreographic programming* [4, 5, 14, 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 [20].

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 $r, s, \ldots$, 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 $x, y, \ldots$. We also assume a set of boolean expressions ranged over by b.

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

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

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 **1** defines a DIOC that can only terminate. **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 **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 **I**. A global state Σ is a map that defines the value v of each variable x in a given role r, namely $\Sigma : \mathit{Roles} \times \mathit{Var} \to \mathit{Val}$. The local state of role r is $\Sigma_r : \mathit{Var} \to \mathit{Val}$ and it verifies $\forall x \in \mathit{Var} : \ \Sigma(r,x) = \Sigma_r(x)$. 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`. **I** denotes a set of updates, i.e., DIOCs that may replace a scope. **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 communication 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 accepted, 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 payment 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

$$\mathsf{transI}(o^{?} : r_1(e) \to r_2(x)) = \mathsf{transF}(o^{?} : r_1(e) \to r_2(x)) = \{r_1 \to r_2\}$$
$$\mathsf{transI}(x@r = e) = \mathsf{transF}(x@r = e) = \{r \to r\}$$
$$\mathsf{transI}(\mathbf{1}) = \mathsf{transI}(\mathbf{0}) = \mathsf{transF}(\mathbf{1}) = \mathsf{transF}(\mathbf{0}) = \emptyset$$
$$\mathsf{transI}(\mathcal{I}|\mathcal{I}') = \mathsf{transI}(\mathcal{I}) \cup \mathsf{transI}(\mathcal{I}') \qquad \mathsf{transF}(\mathcal{I}|\mathcal{I}') = \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}')$$
$$\mathsf{transI}(\mathcal{I};\mathcal{I}') = \begin{cases} \mathsf{transI}(\mathcal{I}') & \text{if } \mathsf{transI}(\mathcal{I}) = \emptyset \\ \mathsf{transI}(\mathcal{I}) & \text{otherwise} \end{cases} \qquad \mathsf{transF}(\mathcal{I};\mathcal{I}') = \begin{cases} \mathsf{transF}(\mathcal{I}) & \text{if } \mathsf{transF}(\mathcal{I}') = \emptyset \\ \mathsf{transF}(\mathcal{I}') & \text{otherwise} \end{cases}$$
$$\mathsf{transI}(\texttt{if } b@r \ \{\mathcal{I}\} \ \texttt{else} \ \{\mathcal{I}'\}) = \mathsf{transI}(\texttt{while } b@r \ \{\mathcal{I}\}) = \{r \to r\}$$
$$\mathsf{transF}(\texttt{if } b@r \ \{\mathcal{I}\} \ \texttt{else} \ \{\mathcal{I}'\}) = \begin{cases} \{r \to r\} & \text{if } \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}') = \emptyset \\ \mathsf{transF}(\mathcal{I}) \cup \mathsf{transF}(\mathcal{I}') & \text{otherwise} \end{cases}$$
$$\mathsf{transF}(\texttt{while } b@r \ \{\mathcal{I}\}) = \begin{cases} \{r \to r\} & \text{if } \mathsf{transF}(\mathcal{I}) = \emptyset \\ \mathsf{transF}(\mathcal{I}) & \text{otherwise} \end{cases}$$
$$\mathsf{transI}(\texttt{scope } @r \ \{\mathcal{I}\}) = \{r \to r\}$$
$$\mathsf{transF}(\texttt{scope } @r \ \{\mathcal{I}\}) = \begin{cases} \{r \to r\} & \text{if } \mathsf{roles}(\mathcal{I}) \subseteq \{r\} \\ \bigcup_{r' \in \mathsf{roles}(\mathcal{I}) \setminus \{r\}} \{r' \to r\} & \text{otherwise} \end{cases}$$

Table 1. Auxiliary functions transI 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 transI 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 transI and transF are $\{r_1 \to r_2\}$. For conditional, $\mathsf{transI}(\texttt{if } b@r \ \{\mathcal{I}\} \ \texttt{else} \ \{\mathcal{I}'\}) = \{r \to r\}$ since the first action executed is the evaluation of the guard by role r. The set $\mathsf{transF}(\texttt{if } b@r \ \{\mathcal{I}\} \ \texttt{else} \ \{\mathcal{I}'\})$ 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 [20].

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

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

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

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

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

[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 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 [15]. 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 [20] (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.

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 `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* $o^* : r_1(v) \rightarrow r_2(x)$, *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 $\mathcal{N}, \mathcal{N}', \ldots$, 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 $\langle \mathbf{I}, \mathcal{N} \rangle$.

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

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

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:*

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

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

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

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

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

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

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

$$\begin{cases} \texttt{if } b \ \{(\mathit{\Pi}_{r' \in \mathsf{roles}(\mathcal{I}, \mathcal{I}') \setminus \{r\}} \ o_n^* : \mathit{true} \texttt{ to } r'); \ \pi(\mathcal{I}, s)\} \\ \quad \texttt{else } \{(\mathit{\Pi}_{r' \in \mathsf{roles}(\mathcal{I}, \mathcal{I}') \setminus \{r\}} \ o_n^* : \mathit{false} \texttt{ to } r'); \ \pi(\mathcal{I}', s)\} & \text{if } s = r \\ o_n^* : x_n \texttt{ from } r; \ \texttt{if } x_n \ \{\pi(\mathcal{I}, s)\} \texttt{ else } \{\pi(\mathcal{I}', s)\} & \text{if } r \in \mathsf{roles}(\mathcal{I}, \mathcal{I}') \setminus \{s\} \\ \mathbf{1} & \text{otherwise} \end{cases}$$

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

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

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

Table 3. Process-projection function π.

a loop $n : \texttt{while } b\texttt{@}r \ \{\mathcal{I}\}$ 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 $\texttt{if } b\texttt{@}r \ \{\mathcal{I}\} \texttt{ else } \{\mathcal{I}'\}$ construct are needed unless one requires that $r \in \{r_1, r_2\}$ for each $r_1 \to r_2 \in \mathsf{transl}(\mathcal{I}) \cup \mathsf{transl}(\mathcal{I}')$. 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] $(o^? : X \texttt{ to } r', \Gamma)_r \xrightarrow{\overline{o^?}\langle X\rangle @ r' : r} (\mathbf{1}, \Gamma)_r$

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

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

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

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

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

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

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 [20] 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 [20] (excerpt in Table 4). Symmetric rules for parallel composition have been omitted.*

$$\text{[Lift]}\quad \frac{\mathcal{N} \xrightarrow{\delta} \mathcal{N}' \quad \delta \neq \mathcal{I}}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\delta} \langle \mathbf{I}, \mathcal{N}' \rangle} \qquad \text{[Lift-Up]}\quad \frac{\mathcal{N} \xrightarrow{\mathcal{I}} \mathcal{N}' \quad \mathcal{I} \text{ connected} \quad \mathcal{I} \in \mathbf{I}}{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\mathcal{I}} \langle \mathbf{I}, \mathcal{N}' \rangle} \qquad \text{[Change-Updates]}\quad \langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\mathbf{I}'} \langle \mathbf{I}', \mathcal{N} \rangle$$

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

$$\text{[Synch-Up]}\quad \frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\overline{o^?}\langle X \rangle @ r_2 : r_1} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{o^?(\leftarrow X) @ r_1 : r_2} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{o^? : r_1(X) \to r_2()} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle}$$

$$\text{[Ext-Parallel]}\quad \frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \rangle \qquad \eta \neq \surd}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{\eta} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}'' \rangle} \qquad \text{[Ext-Par-End]}\quad \frac{\langle \mathbf{I}, \mathcal{N} \rangle \xrightarrow{\surd} \langle \mathbf{I}, \mathcal{N}' \rangle \quad \langle \mathbf{I}, \mathcal{N}'' \rangle \xrightarrow{\surd} \langle \mathbf{I}, \mathcal{N}''' \rangle}{\langle \mathbf{I}, \mathcal{N} \parallel \mathcal{N}'' \rangle \xrightarrow{\surd} \langle \mathbf{I}, \mathcal{N}' \parallel \mathcal{N}''' \rangle}$$

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 $\mathsf{freshIndex}(\mathcal{I}, n)$, 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, $\mathsf{freshIndex}(\mathcal{I}, n)$ 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 and in the set of currently available updates $\mathbf{I}$. 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 $o^? : r_1(v) \to r_2(x)$ 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 $o^? : r_1(X) \to r_2()$ 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* $\eta_i \in \{\tau, o^? : r_1(v) \to r_2(x), o^* : r_1(X) \to r_2(), \surd, \mathcal{I}, \mathsf{no\text{-}up}, \mathbf{I}\}$ *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* $o^* : r_1(v) \to r_2(x)$ *or* $o^* : r_1(X) \to r_2()$*, 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.

In the companion technical report [20] one can find 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

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 the technical report [20]) 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* $\surd$ *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 $o^? : x$ `from` r_1 in role r_2 may interact with the sends of the form $o^? : e$ `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* $\langle \mathbf{I}, \mathsf{proj}(\mathcal{I}, \Sigma) \rangle \xrightarrow{\mu_1} \cdots \xrightarrow{\surd} \langle \mathbf{I}', \mathcal{N} \rangle$*, then* $\mathcal{N}$ *contains no sends.*

5 Related works and discussion

This paper presents an approach for the dynamic update of distributed applications. 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, 12], 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 [9], which uses types to ensure safe adaptation. However, [9] allows updates only when no session is active, while we change the behaviour of running DIOCs. Our work shares with [17] the interest in choreographies composition. However, [17] uses multiparty session types and 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 [10, 16]. 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 adaptation mechanism described in [13]. While applications in [13] 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 [19] and dynamic [23] Aspect-Oriented Programming (AOP) and with Context-Oriented Programming (COP) [11]. 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, 12]. 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 multiparty sessions. *Service Oriented Computing and Applications*, pages 1–20, 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. C. Di Giusto and J. A. Pérez. Disciplined structured communications with consistent runtime adaptation. In *SAC*, pages 1913–1918. ACM, 2013.
10. C. Ghezzi, M. Pradella, and G. Salvaneschi. An evaluation of the adaptation capabilities in programming languages. In *SEAMS*, pages 50–59. ACM, 2011.
11. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
13. 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.
14. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM*, pages 323–332. IEEE, 2008.
15. I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In *WWV*, volume 123, pages 34–48. EPTCS, 2013.
16. L. A. F. Leite et al. A systematic literature review of service choreography adaptation. *Service Oriented Computing and Applications*, 7(3):199–216, 2013.
17. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
18. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.
19. R. Pawlak et al. JAC: an aspect-based distributed dynamic framework. *Softw., Pract. Exper.*, 34(12):1119–1148, 2004.
20. M. D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies - safe runtime updates of distributed applications. Technical report, 2014. `http://arxiv.org/abs/1407.0970`.
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.