

Choreographic Programming of Isolated Transactions

Ton Smeele

Open University of the Netherlands
Heerlen, the Netherlands

Sung-Shik Jongmans

Open University of the Netherlands
Heerlen, the Netherlands
Centrum Wiskunde & Informatica (CWI)
Amsterdam, the Netherlands
ssj@ou.nl

Implementing distributed systems is hard; choreographic programming aims to make it easier. In this paper, we present the design of a new choreographic programming language that supports isolated transactions among overlapping sets of processes. The first idea is to track for every variable which processes are permitted to use it. The second idea is to use model checking to prove isolation.

1 Introduction

1.1 Background: Choreographic Programming

Implementing distributed systems is hard; *choreographic programming* aims to make it easier [8, 10, 37]. Figure 1 shows the idea.

Initially, a distributed system is written as a *global program* G (“the choreography”). It implements the behaviour of all processes collectively, in a sequential programming style (easy to write, but hard to run as a distributed system). For instance, the following global program implements a distributed system in which, first, a data object is communicated from Alice to Bob, and second, its hash.

$$G_{ab} = (a.\text{foo} \rightarrow b.x) ; (a.\text{hash} := \text{md5}(\text{foo})) ; (a.\text{hash} \rightarrow b.y)$$

Here, $p.e \rightarrow q.y$ and $q.y := e$ express inter-process *communication* and intra-process *computation*. Communication $p.e \rightarrow q.y$ implements the output of the value of expression e at process p and the corresponding input into variable y at process q ; the transport is asynchronous, reliable, and FIFO. Computation $q.y := e$ implements the storage of the value of expression e in variable y at process q .

Subsequently, the distributed system is run as a family of *local programs* L_1, \dots, L_n , automatically extracted from the global program through *projection*. The local programs implement the behaviour of each process individually, in a parallel programming style (easy to run as a distributed system, but hard to write). For instance, the following local programs implement Alice and Bob:

$$L_a = (ab!\text{foo}) ; (a.\text{hash} := \text{md5}(\text{foo})) ; (ab!\text{hash}) \quad L_b = (ab?x) ; (ab?y)$$

Here, *send* $pq!e$ and *receive* $pq?y$ implement an output and an input through the channel from p to q .

The keystone assurance of choreographic programming is *operational equivalence*: methodically, a global program and its family of local programs are assured to have the same behaviour. To prove properties of families of local programs, operational equivalence allows us to prove them of global programs instead. This is typically simpler. A premier example of such a property is *absence of deadlocks*.

Choreographic programming originated with Carbone et al. [7, 8] (using binary session types [31]) and with Carbone and Montesi [10, 37] (using multiparty session types [32]); substantial progress has been made since. Montesi and Yoshida developed a theory of compositional choreographic programming

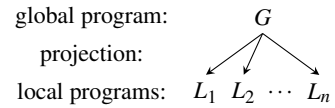


Figure 1: Method

that supports open distributed systems [38]; Carbone et al. studied connections between choreographic programming and linear logic [6, 11]; Dalla Preda et al. combined choreographic programming with dynamic adaptation [39–41]; Cruz-Filipe and Montesi developed a minimal Turing-complete language of global programs [21]; Cruz-Filipe et al. and Kjær et al. presented techniques to extract global programs from families of local programs [17, 35]; Giallorenzo et al. studied a correspondence between choreographic programming and multitier languages [27]; Jongmans and Van den Bos combined choreographic programming with deductive verification [34]; Hirsch and Garg and Cruz-Filipe et al. developed functional choreographic programming languages [16, 30]. Other work includes results on case studies [18], procedural abstractions [20], asynchronous communication [19], polyadic communication [22, 29], implementability [26], and formalisation/mechanisation in Coq [23, 24, 30]. These theoretical developments are supported in practice by several tools [4, 10, 27, 40, 41].

1.2 Open Problem: Isolated Transactions

Suppose we need to implement a distributed system that fulfils the following requirements:

1. A data object and its hash are communicated from both Alice and Carol, in parallel, to Bob.
2. Either Alice’s data object and its hash are eventually stored at Bob, or Carol’s (but no mixture).

Requirement 1 can readily be fulfilled in a choreographic programming language with parallel composition (free interleaving), as demonstrated in the following global program:

$$G_{acb}^{v1} = G_{ab}^{s1.1} \parallel G_{cb} \quad G_{cb} = (c.\text{"bar"} \rightarrow b.x) ; (c.\text{hash} := \text{md5}(\text{"bar"})) ; (c.\text{hash} \rightarrow b.y)$$

In contrast, requirement 2 cannot be fulfilled in any choreographic programming language that we know of (i.e., none of the choreographic programming languages cited in §1.1 seem to be capable of it). What is needed, is a mechanism to run G_{ab} and G_{cb} as isolated *transactions*.

One possibility is to enrich the language with the standard non-deterministic choice operator $+$. In that case, the system can be implemented as $(G_{ab} ; G_{cb}) + (G_{cb} ; G_{ab})$. However, such an approach, in which parallel compositions are explicitly expanded into choices, generally leads to exponentially sized global programs (in the number of transactions), while obscuring the intention of the system. For instance, if Dave were added as a third client of Bob, we need to write the following global program:

$$\begin{aligned} G_{abcd}^{v1} = & (G_{ab} ; ((G_{cb} ; G_{db}) + (G_{db} ; G_{cb}))) + & G_{db} = & (d.\text{"baz"} \rightarrow b.x) ; \\ & (G_{cb} ; ((G_{ab} ; G_{db}) + (G_{db} ; G_{ab}))) + & & (d.\text{hash} := \text{md5}(\text{"baz"})) ; \\ & (G_{db} ; ((G_{ab} ; G_{cb}) + (G_{cb} ; G_{ab}))) & & (d.\text{hash} \rightarrow b.y) \end{aligned}$$

Moreover, if we want to allow *independent segments* of transactions, which use disjoint sets of variables, to overlap to improve performance (i.e., their interleaved execution would not break isolation), then programmability is further complicated with the non-deterministic choice approach.

To avoid these issues, we propose a more fine-grained approach in this paper that supports eventual consistency while allowing for interleaved execution of isolated transactions. Instead of manually implementing isolated transactions by enumerating admissible sequences of communications, in our approach, isolation emerges out of explicit programming language support.

1.3 Contributions of This Paper

We present the design of a new choreographic programming language that supports isolated transactions.

The first idea is to track for each variable which processes are permitted to use it. Initially, each process is permitted to use each variable. Subsequently, process p can *acquire* exclusive permission

to use variable y of process q . When granted, each usage of y by $\text{not-}p$ is blocked until p releases its exclusive permission. Management of usage permissions is transparent to the programmer; it is a feature of the programming language. The following global programs demonstrate the syntax and fulfil requirement 2 in §1.2:

$$G_{acb}^{v2} = ((a \text{ acq } b.x) ; G_{ab}^{\S1.1} ; (a \text{ rel } b.x)) \parallel ((c \text{ acq } b.x) ; G_{cb}^{\S1.2} ; (c \text{ rel } b.x))$$

$$G_{acdb}^{v2} = G_{acb}^{v2} \parallel ((d \text{ acq } b.x) ; G_{db}^{\S1.2} ; (d \text{ rel } b.x))$$

We note that G_{acdb}^{v2} is *compositionally constructed* out of G_{acb}^{v2} , without the need to refer to sub-programs G_{ab} and G_{cb} ; this is not possible when parallel compositions are explicitly expanded into choices.

Thus, the idea of tracking usage permissions—and blocking those usages that are forbidden—enables the programmer to write more compact global programs, intended to better preserve the intention of the system. However:

- This feature does not guarantee isolation by itself; it is just a means to achieve it. In other words, a separate mechanism is still needed to check isolation and guarantee it is preserved by projection.
- “Blocking those usages that are forbidden” also has an adverse side-effect: processes that compete to acquire permission to use the same variables can deadlock. For instance, the following global program implements a system in which Alice tries to acquire permission to use variables x and y of Bob, while Carol tries to acquire permission to use the same variables, but in reverse:

$$(a \text{ acq } b.x ; a \text{ acq } b.y ; \dots) \parallel (c \text{ acq } b.y ; c \text{ acq } b.x ; \dots)$$

A deadlock arises when Alice acquires permission to use x , while Carol acquires permission to use y , so neither one of them can acquire permission to use a second variable.¹

To address these points, the second idea of this paper is to specify properties, such as isolation and absence of deadlocks, in temporal logic and use model checking to prove that they are satisfied. We believe this combination with choreographic programming is new.

2 The Design

We define a language in which both global programs and families of local programs can be expressed. Figure 2 shows the design. It has four layers: every *system* is defined in terms of *programs* (either a single global one, or multiple local ones), *stores* (one for every process), and *channels* (one between every pair of processes); every program, store, or channel is defined in terms of *actions*, process/channel *names*, and *data*; every action is itself defined in terms of names and data, too.

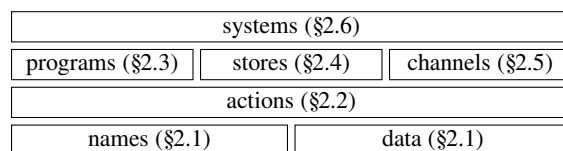


Figure 2: Design

2.1 Names and Data

First, we define: the syntax of names (Definition 1); the syntax of data (Definition 2). As the topic of interest is “processes that communicate”, instead of “data that are communicated”, we omit most details.

¹We note that this is a different source of deadlock than the *communication deadlocks* that choreographic programming traditionally avoids (i.e., waiting for a message that is never sent).

Definition 1. Let $\mathbb{R} = \{a, b, c, \dots\}$ denote the set of *process names*, ranged over by p, q, r . Let $\mathbb{R} \times \mathbb{R} \setminus \{(r, r) \mid r \in \mathbb{R}\}$ denote the set of *channel names*. \square

Definition 2. Let $\mathbb{X} = \{_, x, y, z, \dots\}$ denote the set of *variables*, ranged over by x, y, z . Let $\mathbb{V} = \{\text{unit}, \text{true}, \text{false}, 0, 1, 2, \dots, \text{acq}, \text{rel}\}$ denote the set of *values*, ranged over by u, v, w . Let \mathbb{E} denote the set of *expressions*, ranged over by E ; it is defined as follows:

$$E ::= x \mid u \mid E_1 == E_2 \mid \sim E \mid E_1 \&\& E_2 \mid E_1 + E_2 \mid \dots \quad \square$$

Symbol $_$ is a special variable that loses all data written to it, similar to `/dev/null` in Unix. Symbols `acq` and `rel` are special values to control usage permissions of variables (§2.4).

2.2 Actions

Next, we define: the syntax of actions that processes can execute (Definition 3); functions to retrieve the “subject” and the “object” of an action (Definition 4). The subject is the process that executes an action; the object is the channel through which an action is executed, if any.

Definition 3. Let \mathbb{A} denote the set of *actions*, ranged over by α ; it is defined as follows:

$$\alpha ::= p.E \mid q.y := E \mid pq!E \mid pq?E \mid \tau \quad \square$$

Action $p.E$ implements a *test* of expression E at process p . Action $q.y := E$ implements an *assignment* of the value of expression E to variable y at process q . Actions $pq!E$ and $pq?E$ implement an *asynchronous send* and *receive* of the value of expression E from process p to process q . Action τ implements *idling*.

Definition 4. Let $\text{subj}(\alpha)$ and $\text{obj}(\alpha)$ denote the *subject* and the *object* of α ; they are defined as follows:

$$\begin{array}{lll} \text{subj}(p.E) = p & \text{subj}(q.y := E) = q & \text{obj}(pq!E) = pq \\ \text{subj}(pq!E) = p & \text{subj}(pq?y) = q & \text{obj}(pq?E) = pq \end{array} \quad \square$$

2.3 Programs

Next, we define: the syntax of programs (Definition 5); a function to extract local programs from a global program (Definition 6); the operational semantics of programs (Definition 7).

Definition 5. Let \mathbb{P} denote the set of *programs*, ranged over by P, G, L ; it is defined as follows:

$$P ::= \mathbf{1} \mid \alpha \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P_1 ; P_2 \quad \square$$

Program $\mathbf{1}$ implements an *empty execution*. Program $P_1 + P_2$ implements a *choice* between P_1 and P_2 . Program $P_1 \parallel P_2$ implements an *interleaving* of P_1 and P_2 . Program $P_1 ; P_2$ implements a *sequence* of P_1 and P_2 . Furthermore, we use the following shorthand notation:

$$\begin{array}{l} p.E \rightarrow q.y \text{ instead of } pq!E ; pq?y \\ p \mathbf{acq} q.y \text{ instead of } (p.\mathbf{acq} \rightarrow q.y) ; (q.\mathbf{unit} \rightarrow p._) \\ p \mathbf{acq} q.[y_1, \dots, y_n] \text{ instead of } p \mathbf{acq} q.y_1 ; \dots ; p \mathbf{acq} q.y_n \\ p \mathbf{rel} q.y \text{ instead of } p.\mathbf{rel} \rightarrow q.y \\ p \mathbf{rel} q.[y_1, \dots, y_n] \text{ instead of } p \mathbf{rel} q.y_1 ; \dots ; p \mathbf{rel} q.y_n \\ \mathbf{if} p.e P_1 P_2 \text{ instead of } (p.E ; P_1) + (p.\sim E ; P_2) \end{array}$$

A program is *global* if at least two subjects occur in it; it is *local* if it at most one subject occurs in it. A local program for process r can be extracted from global program G through projection. The idea is to replace every action in G of which r is not the subject with τ .

Definition 6. Let $P \upharpoonright r$ denote the *projection* of P onto r ; it is induced by the following equations:

$$\begin{array}{c}
\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}} \quad \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 + P_2 \xrightarrow{\alpha} P'_1} \quad \frac{P_2 \xrightarrow{\alpha} P'_2}{P_1 + P_2 \xrightarrow{\alpha} P'_2} \quad \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 \parallel P_2 \xrightarrow{\alpha} P'_1 \parallel P_2} \quad \frac{P_2 \xrightarrow{\alpha} P'_2}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1 \parallel P'_2} \\
\\
\frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 ; P_2 \xrightarrow{\alpha} P'_1 ; P_2} \quad \frac{\text{subj}(\alpha) \notin \{\text{subj}(\hat{\alpha}) \mid P_1 \rightarrow \dots \rightarrow \hat{\alpha}\}}{P_1 ; P_2 \xrightarrow{\alpha} P'_2}
\end{array}$$

(a) Programs. Let $\rightarrow \dots \rightarrow$ denote a sequence of 0-or-more reductions.

$$\begin{array}{c}
\frac{S[[E]]_p = \text{true}}{S \xrightarrow[p.\text{true}]{p.E} S} \quad \frac{S[[E]]_q = v}{S \xrightarrow[q.y:=v]{q.y:=E} S[y \mapsto v]_q} \quad \frac{S[[E]]_p = u}{S \xrightarrow[pq!u]{pq!E} S} \quad \frac{}{S \xrightarrow[pq?v]{pq?y} S[y \mapsto v]_p} \quad \frac{}{S \xrightarrow{\tau} S}
\end{array}$$

(b) Stores

$$\begin{array}{c}
\frac{}{C \xrightarrow[p.v]{} C} \quad \frac{}{C \xrightarrow[q.y:=v]{} C} \quad \frac{|\vec{v}| < n}{(\vec{v}, n) \xrightarrow[pq!u]{} (u \cdot \vec{v}, n)} \quad \frac{}{(\vec{u} \cdot v, n) \xrightarrow[pq?v]{} (\vec{u}, n)} \quad \frac{}{C \xrightarrow{\tau} C}
\end{array}$$

(c) Channels

$$\begin{array}{c}
\frac{P \xrightarrow{\alpha} P'}{\{P\} \cup \mathcal{P} \xrightarrow{\alpha} \{P'\} \cup \mathcal{P}} \quad \frac{S \xrightarrow[\alpha]{\alpha} S' \quad \text{subj}(\alpha) = r}{\{\text{subj}(\alpha) \mapsto S\} \cup S \xrightarrow{\alpha} \{\text{subj}(\alpha) \mapsto S'\} \cup S} \\
\\
\frac{C \xrightarrow[\alpha]{} C' \quad \text{obj}(\alpha) = pq}{\{pq \mapsto C\} \cup C \xrightarrow{\alpha} \{pq \mapsto C'\} \cup C} \quad \frac{\mathcal{P} \xrightarrow{\alpha} \mathcal{P}' \quad S \xrightarrow[\alpha]{\alpha} S' \quad C \xrightarrow[\alpha]{} C'}{(\mathcal{P}, S, C) \xrightarrow{\alpha} (\mathcal{P}', S', C')}
\end{array}$$

(d) Systems

Figure 3: Operational semantics

$$\begin{array}{l}
\alpha \upharpoonright \text{subj}(\alpha) = \alpha \qquad \mathbf{1} \upharpoonright r = \mathbf{1} \\
\alpha \upharpoonright r = \tau \quad \text{if: } r \neq \text{subj}(\alpha) \quad P_1 \circ P_2 \upharpoonright r = (P_1 \upharpoonright r) \circ (P_2 \upharpoonright r) \quad \text{if: } \circ \in \{+, \parallel, ;\} \quad \square
\end{array}$$

We define the operational semantics of programs through a labelled reduction relation.

Definition 7. Let $P \xrightarrow{\alpha} P'$ denote *reduction* from P to P' with α ; it is defined in Figure 3a. \square

Most rules are standard. The only special rule is the second rule for sequencing: it allows sequences of actions to be executed *out-of-order*, so long as they are executed at different processes (i.e., they are independent; insisting on a sequential order would be unreasonable in a parallel environment). That is, the left premise of the rule entails that the subject of α does not occur in P_1 (cf. the operational semantics of global multiparty session types). For instance, in $a.x:=5 ; b.y:=6$, the assignments at Alice and Bob may be executed out-of-order. In contrast, in $a.x:=5 ; a.x+1 \rightarrow b.y$, the assignment and the communication must be executed in-order.

2.4 Stores

Next, we define: the syntax of stores (Definition 8); functions to read expressions from a store and write values to it (Definition 9); the operational semantics of stores (Definition 10).

Definition 8. Let $\mathbb{S} = (\mathbb{X} \setminus \{-\}) \rightarrow (\mathbb{V} \times 2^{\mathbb{R}})$ denote the set of *stores*, ranged over by S . \square

Storage $S(x) = (u, R)$ means that variable x has value u , and that the processes in R are permitted to use it. Typically, $R \in \{\mathbb{R}\} \cup \{\{r\} \mid r \in \mathbb{R}\}$: either every process is permitted to use x (if $R = \mathbb{R}$), or only one process (if $R = \{r\}$ for some $r \in \mathbb{R}$). Every process has its own store, but through communications, other processes can use it, too.

Definition 9. Let $S[[E]]_r$ and $S[y \mapsto v]_r$ denote the *read* of E in S by r and the *write* of v to y in S by r ; they are defined as follows:

$$\begin{array}{lll}
S[[x]]_r & = u & \mathbf{if: } S(x) = (u, R) \mathbf{ and } r \in R & S[[E_1 == E_2]]_r = \dots \\
S[[u]]_r & = u & & S[[\sim E]]_r = \dots \\
S[[_ \mapsto v]]_r & = S & & S[[E_1 \&\& E_2]]_r = \dots \\
S[y \mapsto v]_r & = \{x \mapsto S(x) \mid x \neq y\} \cup \begin{cases} \{y \mapsto (v, R)\} & \mathbf{if: } \text{acq} \neq v \neq \text{rel} \\ \{y \mapsto (u, \{r\})\} & \mathbf{if: } \text{acq} = v \neq \text{rel} \\ \{y \mapsto (u, \mathbb{R})\} & \mathbf{if: } \text{acq} \neq v = \text{rel} \end{cases} & & S[[E_1 + E_2]]_r = \dots \\
& & \mathbf{if: } y \neq _ \mathbf{ and } S(y) = (u, R) \mathbf{ and } r \in R & \begin{array}{l} \vdots \\ \vdots \\ \vdots \end{array}
\end{array}
\quad \square$$

Writes $S[y \mapsto \text{acq}]_r$ and $S[y \mapsto \text{rel}]_r$ mean that process r tries to acquire or release exclusive permission to use y , without changing the value; it succeeds only if r already has permission (possibly non-exclusive).

The crux of the definition is that $S[[E]]_r$ and $S[y \mapsto v]_r$ are undefined when r is not permitted to use a variable that occurs in E or y . Such undefinedness is leveraged in the operational semantics of stores (next definition). We note that $S[[E]]_r$ is also undefined when operations are performed on sub-expressions of incompatible types. For instance, $S[[5 + \text{true}]]_r$ is undefined. A type system can be used to catch such errors statically; this is orthogonal to the aim of this paper.

We define the operational semantics of stores through a labelled reduction relation. Every reduction has two labels: an action (written above the arrow) and the ‘‘ground’’ version of the action (written below). In the ground version, every expression is replaced by its value, if any.

Definition 10. Let $S \xrightarrow[\alpha]{\alpha'} S'$ denote *reduction* from S to S' with α and α' ; it is defined in Figure 3b. \square

The first rule states that a test $p.E$ is executed on a store by reading E , if the value of E is `true`, and if p has enough permissions. The second rule states that an assignment $q.y := E$ is executed by reading E , and by writing the value of E to y , if q has enough permissions. The third rule states that a send $pq!E$ is executed by reading E , if p has enough permissions. The fourth rule states that a receive $pq?y$ and its ground version $pq?v$ are executed by writing v to y , if p has permission to use y (not q ; essentially, we treat receives as remote assignments). If a process does not have enough permissions for a rule to be applicable, the store cannot reduce, so the action is blocked.

2.5 Channels

Next, we define: the syntax of channels (Definition 11); the operational semantics (Definition 12). Henceforth, we write \vec{u} for a list of values, and we write $v \cdot \vec{u}$ and $\vec{u} \cdot v$ for prefixing and suffixing.

Definition 11. Let $\mathbb{C} = \mathbb{V}^* \times \{0, 1, 2, \dots, \infty\}$ denote a set of *channels*, ranged over by C . \square

Channel (\vec{u}, n) means that its n -capacity buffer contains the values in \vec{u} ; the buffer is reliable and FIFO.

We define the operational semantics of channels through a labelled reduction relation. As channels contain values, every reduction has one label: a ground action (written below the arrow).

Definition 12. Let $C \xrightarrow[\alpha]{} C'$ denote *reduction* from C to C' with α ; it is defined in Figure 3c. \square

The first and second rule state that a test and an assignment are executed on a channel without really using it. The third rule states that a send is executed by enqueueing a value to the buffer, if it is not full. The fourth rule states that a receive is executed by dequeueing a value from the buffer, if it is not empty. Henceforth, we omit reduction labels when they do not matter.

2.6 Systems

Last, we define: the syntax of systems (Definition 13); the operational semantics (Definition 14); operational equivalence (Definition 15)

Definition 13. Let $\mathbb{P} = 2^{\mathbb{P}} \setminus \{\emptyset\}$ denote the set of (non-empty) *sets of programs*, ranged over by \mathcal{P} . Let $\mathbb{S} = \mathbb{R} \rightarrow \mathbb{S}$ denote the set of *families of stores*, ranged over by \mathcal{S} . Let $\mathbb{C} = \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{C}$ denote the set of *families of channels*, ranged over by \mathcal{C} . Let $\mathbb{P} \times \mathbb{S} \times \mathbb{C}$ denote the set of *systems*, ranged over by \mathcal{D} . \square

System $(\mathcal{P}, \mathcal{S}, \mathcal{C})$ means that the program(s) in \mathcal{P} , the stores in \mathcal{S} , and the channels in \mathcal{C} are executed together. It is well-formed if there exists a set of processes $R = \{r_1, \dots, r_n\}$ such that the domain of \mathcal{S} is R (every process has a store), and the domain of \mathcal{C} is $R \times R$ (every pair of processes has a channel), and:

$$\mathcal{P} \in \{ \{P\} \mid P \text{ is global and every subject that occurs in } P \text{ occurs in } R \} \cup \\ \{ \{P_1, \dots, P_n\} \mid \text{for each } 1 \leq i \leq n, P_{r_i} \text{ is local and every subject that occurs in } P_{r_i} \text{ is } r_i \}$$

We define the operational semantics of systems through a labelled reduction relation.

Definition 14. Let $(\mathcal{P}, \mathcal{S}, \mathcal{C}) \xrightarrow[\alpha]{} (\mathcal{P}, \mathcal{S}, \mathcal{C})'$ denote *reduction* from $(\mathcal{P}, \mathcal{S}, \mathcal{C})$ to $(\mathcal{P}, \mathcal{S}, \mathcal{C})'$ with α and $\underline{\alpha}$; it is defined in Figure 3d. \square

The first, second, and third rule lift reduction from individual programs, stores, and channels to sets of programs, families of stores, and families of channels. The fourth rule connects them together.

Two systems are operationally equivalent if they have the same behaviour. We formalise “having the same behaviour” in terms of *branching bisimilarity* [28] (in contrast to trace equivalence as usual), because: it is insensitive to idling; it preserves the validity of formulas in many temporal logics (including LTL, CTL, CTL*, and μ -calculus, subject to conditions), which we require to specify properties of global programs. Two systems (resp. processes, stores, channels, sets of processes, families of stores, families of channels) are branching bisimilar iff they can repeatedly mimic each other’s reductions, modulo idling.

Definition 15. Let $\{\approx_1, \approx_2, \dots\}$ denote the set of *branching bisimulations*, ranged over by \approx ; it is defined as follows, coinductively:

- for each $D_1 \xrightarrow{\tau}^* D_1^\dagger \xrightarrow[\alpha]{} D_1^\ddagger \xrightarrow{\tau}^* D_1'$, for some $D_2 \xrightarrow{\tau}^* D_2^\dagger \xrightarrow[\alpha]{} D_2^\ddagger \xrightarrow{\tau}^* D_2'$, $D_1^\dagger \approx D_2^\dagger$, $D_1^\ddagger \approx D_2^\ddagger$, $D_1' \approx D_2'$
 - for each $D_2 \xrightarrow{\tau}^* D_2^\dagger \xrightarrow[\alpha]{} D_2^\ddagger \xrightarrow{\tau}^* D_2'$, for some $D_1 \xrightarrow{\tau}^* D_1^\dagger \xrightarrow[\alpha]{} D_1^\ddagger \xrightarrow{\tau}^* D_1'$, $D_1^\dagger \approx D_2^\dagger$, $D_1^\ddagger \approx D_2^\ddagger$, $D_1' \approx D_2'$
-
- $$D_1 \approx D_2$$

Let $\equiv = \approx_1 \cup \approx_2 \cup \dots$ denote *operational equivalence* (i.e., the largest branching bisimulation). \square

The following proposition states that operational equivalence of sets of programs implies that of the systems they constitute. Specifically, if P is a global program, and if $\{P\} \equiv \{P \upharpoonright r \mid r \text{ is a subject of } P\}$, then the local programs extracted from P have the same behaviour as P in *any* initial stores and channels. In the absence of loops, as in this paper, checking $\mathcal{P}_1 \equiv \mathcal{P}_2$ is clearly decidable; in the presence of loops, it is not. We leave decidable approximations of \equiv (e.g., well-formedness conditions on the syntax of choices, as usual) for future work, when we extend our work with loops.

Proposition 1. For all \mathcal{S}, \mathcal{C} , if $\mathcal{P}_1 \equiv \mathcal{P}_2$, then $(\mathcal{P}_1, \mathcal{S}, \mathcal{C}) \equiv (\mathcal{P}_2, \mathcal{S}, \mathcal{C})$. \square

2.7 Properties

To prove properties, we adopt a state-based temporal logic in the style of CTL [25]. We are primarily interested in two classes of properties (although other classes may be specified, too): isolation of transactions and absence of deadlock; our logic has special predicates to formulate such properties. The need to explicitly prove absence of deadlock arises from the fact that systems in this paper are not deadlock-free by construction. For instance, any system that consists of the following program can deadlock (elaboration of the last example in §1.3):

$$G_{acb}^{v3} = ((a \text{ acq } b.[x, y]) ; G_{ab}^{\S 1.1} ; (a \text{ rel } b.[x, y])) \parallel ((c \text{ acq } b.[y, x]) ; G_{cb}^{\S 1.2} ; (c \text{ rel } b.[x, y]))$$

The problem is that Alice acquires x and y (in that order), while Carol acquires y and x (in that order).

Definition 16. Let \mathbb{F} denote the set of *formulas*, ranged over by φ ; it is defined as follows:

$$\varphi ::= \top \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \text{EG}(\varphi) \mid \text{EU}(\varphi_1, \varphi_2) \mid p.E \mid \text{AX}_{q,y}(\varphi) \mid \text{dead} \quad \square$$

Formula \top specifies *truth*. Formulas $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ specify *negation* and *conjunction*. Formula $\text{EG}(\varphi)$ specifies that, in some branch, φ is *always* true. Formula $\text{EU}(\varphi_1, \varphi_2)$ specifies that, in some branch, φ_1 is true *until* φ_2 is true. Formula $p.E$ specifies *proposition* E at p . Formula $\text{AX}_{q,y}(\varphi)$ specifies that φ is true *next* if variable y at process q was changed. Formula *dead* specifies the presence of deadlock. Furthermore, we use the following shorthand notation (standard):

$$\begin{array}{ll} \perp \text{ instead of } \neg\top & \text{AG}(\varphi) \text{ instead of } \text{EU}(\top, \neg\varphi) \\ \varphi_1 \vee \varphi_2 \text{ instead of } \neg(\neg\varphi_1 \wedge \neg\varphi_2) & \text{AU}(\varphi_1, \varphi_2) \text{ instead of } \neg(\text{EU}(\neg\varphi_2, \neg(\varphi_1 \vee \varphi_2)) \vee \text{EG}(\neg\varphi_2)) \end{array}$$

Definition 17. Let $\mathcal{D} \models \varphi$ denote *entailment* of φ by \mathcal{D} ; it is defined as follows:

$$\begin{array}{c} \frac{}{\mathcal{D} \models \top} \quad \frac{\mathcal{D} \not\models \varphi}{\mathcal{D} \models \neg\varphi} \quad \frac{\mathcal{D} \models \varphi_1 \quad \mathcal{D} \models \varphi_2}{\mathcal{D} \models \varphi_1 \wedge \varphi_2} \quad \frac{\mathcal{D} \rightarrow \mathcal{D}' \quad \mathcal{D} \models \varphi \quad \mathcal{D}' \models \text{EG}(\varphi)}{\mathcal{D} \models \text{EG}(\varphi)} \\ \\ \frac{\mathcal{D} \models \varphi_2}{\mathcal{D} \models \text{EU}(\varphi_1, \varphi_2)} \quad \frac{\mathcal{D} \rightarrow \mathcal{D}' \quad \mathcal{D} \models \varphi_1 \quad \mathcal{D}' \models \text{EU}(\varphi_1, \varphi_2)}{\mathcal{D} \models \text{EU}(\varphi_1, \varphi_2)} \\ \\ \frac{\mathcal{S} \xrightarrow[p.\text{true}]{p.E} \mathcal{S}}{(\mathcal{P}, \mathcal{S}, \mathcal{C}) \models p.E} \quad \frac{\text{for each } (\mathcal{P}, \mathcal{S}, \mathcal{C}) \rightarrow (\mathcal{P}', \mathcal{S}', \mathcal{C}') \text{ if } \mathcal{S}(q)(y) \neq \mathcal{S}'(q)(y), \text{ then } (\mathcal{P}', \mathcal{S}', \mathcal{C}') \models \varphi}{(\mathcal{P}, \mathcal{S}, \mathcal{C}) \models \text{AX}_{q,y}(\varphi)} \quad \frac{\mathcal{P} \rightarrow (\mathcal{P}, \mathcal{S}, \mathcal{C}) \not\rightarrow}{(\mathcal{P}, \mathcal{S}, \mathcal{C}) \models \text{dead}} \quad \square \end{array}$$

The rules on the first two lines are the standard ones for CTL. The first rule on the third line states that a proposition is true if the corresponding test succeeds. The second rule on the third line states that every reduction that changes variable y at process q must make φ true. The third rule on the third line states that the presence of deadlock is true if the set of programs can reduce, but the system cannot (i.e., program reduction is blocked by stores and/or channels).

In the absence of loops, as in this paper, the model checking problem is decidable: it is straightforward to adapt classical model checking algorithms for CTL (e.g., Clarke et al. [13]) to also support our formulas $p.E$, $AX_{q,y}(\varphi)$, and dead . If a global program G satisfies operational equivalence, then it suffices to model check the system that consists of G instead of model checking the system that consists of G 's projections; the former is generally much more efficient as the state space of G 's projections can be exponentially larger than that of G (due to τ -reductions of the projections).

2.8 Examples

We end this section with some examples. Let:

$$\begin{aligned} \mathcal{S} &= \{a \mapsto \{\text{hash} \mapsto 0\}, b \mapsto \{x \mapsto "", y \mapsto 0\}, c \mapsto \{\text{hash} \mapsto 0\}\} \\ \mathcal{C} &= \{pq \mapsto (\epsilon, \infty) \mid p, q \in \{a, b, c\} \text{ and } p \neq q\} \end{aligned}$$

In words, \mathcal{S} is an initial family of stores (for Alice, Bob, and Carol) in which all variables have default values, while \mathcal{C} is an initial family of empty channels (between Alice, Bob, and Carol). Furthermore, in addition to $G_{\text{acb}}^{\text{v}1}$ in §1.2, $G_{\text{acb}}^{\text{v}2}$ in §1.3, and $G_{\text{acb}}^{\text{v}3}$ in §2.7, let:

$$G_{\text{acb}}^{\text{v}4} = ((a \text{ acq } b.x) ; G_{\text{ab}}^{\text{s}1.1} ; (a \text{ rel } b.x)) \parallel G_{\text{cb}}^{\text{s}1.2} \quad G_{\text{acb}}^{\text{v}5} = (G_{\text{ab}}^{\text{s}1.1} ; G_{\text{cb}}^{\text{s}1.2}) + (G_{\text{cb}}^{\text{s}1.2} ; G_{\text{ab}}^{\text{s}1.1})$$

- Regarding isolation of transactions, the property to be proved can be specified as follows:

$$\varphi = \text{AG}(AX_{b,x}(\text{AU}(AX_{b,x}(\perp) \wedge AX_{b,y}(\perp), AX_{b,y}(b.(\text{md}5(x) == y))))))$$

That is: in all branches, always (AG), if x is changed at Bob ($AX_{b,x}$), it is not changed again ($AX_{b,x}(\perp)$) until y is changed at Bob ($AX_{b,y}$) such that x and y are consistent ($b.(\text{md}5(x) == y)$).

System $(\{G_{\text{acb}}^{\text{v}1}\}, \mathcal{S}, \mathcal{C})$ violates φ , as informally explained in §1.2. System $(\{G_{\text{acb}}^{\text{v}2}\}, \mathcal{S}, \mathcal{C})$ satisfies φ , as Alice and Carol acquire exclusive permission to use x and y at Bob. System $(\{G_{\text{acb}}^{\text{v}3}\}, \mathcal{S}, \mathcal{C})$ also satisfies φ : when the system does deadlock, it does so before x at Bob is changed; when it does not deadlock, Alice and Carol acquire exclusive permission. System $(\{G_{\text{acb}}^{\text{v}4}\}, \mathcal{S}, \mathcal{C})$ violates φ : while $G_{\text{ab}}^{\text{s}1.1}$ runs as an isolated transaction (as Alice does acquire exclusive permission), $G_{\text{cb}}^{\text{s}1.2}$ is not (as Carol does not). System $(\{G_{\text{acb}}^{\text{v}5}\}, \mathcal{S}, \mathcal{C})$ satisfies φ , too, but it violates operational equivalence.

- Regarding absence of deadlocks, the property to be proved can be specified as $\varphi = \text{AG}(\neg \text{dead})$. Systems $(\{G_{\text{acb}}^{\text{v}1}\}, \mathcal{S}, \mathcal{C})$, $(\{G_{\text{acb}}^{\text{v}2}\}, \mathcal{S}, \mathcal{C})$, $(\{G_{\text{acb}}^{\text{v}4}\}, \mathcal{S}, \mathcal{C})$, and $(\{G_{\text{acb}}^{\text{v}5}\}, \mathcal{S}, \mathcal{C})$ satisfy φ . In contrast, system $(\{G_{\text{acb}}^{\text{v}3}\}, \mathcal{S}, \mathcal{C})$ violates φ .

3 Conclusion

3.1 Related Work

Advances in choreographic programming were cited in §1.1. Outside choreographic programming, closest to our work are mechanisms in the literature on session types to assure mutual exclusion. In the literature on *binary* session types, mutual exclusion and related patterns are supported in the work of Balzer et al. [1] (without deadlock freedom) and by Balzer et al. and Kokke et al. [2, 36] (with deadlock freedom) in the form of typing disciplines for linear and shared channels. In the literature *multiparty* session types, mutual exclusion is supported in the work of Voinea et al. [42] in the form of a typing discipline for linear and shared channels in the special case when *multiple processes* together implement *a single role*. More generally, parallel composition has been studied in the context of multiparty session typing in several ways: through static interleaving of types (e.g., [32, 33]); through dynamic interleaving of programs (e.g., [3, 14]); through a combination of those two (e.g., in the form of nesting [9, 12]).

3.2 This Work

We presented the design of a new choreographic programming language that supports isolated transactions among overlapping sets of processes. The first idea was to track for every variable which processes are permitted to use it. The second idea was to use model checking to prove isolation. This paper is our first one in which we pursue these ideas. We believe there is plenty of room to explore alternative designs and/or refine our work as presented. Examples include new primitives in the choreographic programming language to implement programs and new modalities in the temporal logic to specify properties.

3.3 Future Work

On the theoretical side, we see three main avenues. First, we aim to extend the choreographic programming language with primitives that guarantee isolation and absence of deadlocks by construction. One possible design is a primitive of the form “**isolate** P ” that implements P as an isolated transaction. The challenge is to define the operational semantics such that exclusive permission of variables is automatically acquired as late as possible, and released as soon as possible, while avoiding deadlocks (e.g., by imposing a total order on variables). Second, we aim to study an extension of our choreographic programming language with loops. Third, we aim to investigate symbolic methods to prove properties.

On the practical side, we are now developing a proof-of-concept implementation of the design in the form of a compiler from our choreographic programming language to mCRL2 [5, 15]. On input of a global program, the compiler extracts a family of local programs through projection and translates both the global program and its family to mCRL2 specifications. Using the mCRL2 toolset, we can then check properties of the global program (μ -calculus versions of our CTL formulas) and operational equivalence.

References

- [1] Stephanie Balzer & Frank Pfenning (2017): *Manifest sharing with session types*. *Proc. ACM Program. Lang.* 1(ICFP), pp. 37:1–37:29, doi:10.1145/3110281.
- [2] Stephanie Balzer, Bernardo Toninho & Frank Pfenning (2019): *Manifest Deadlock-Freedom for Shared Session Types*. In: *ESOP, Lecture Notes in Computer Science* 11423, Springer, pp. 611–639, doi:10.1007/978-3-030-17184-1_22.
- [3] Lorenzo Bettini, Mario Coppo, Loris D’Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In: *CONCUR, Lecture Notes in Computer Science* 5201, Springer, pp. 418–433, doi:10.1007/978-3-540-85361-9_33.
- [4] Petra van den Bos & Sung-Shik Jongmans (2023): *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*. In: *FM, Lecture Notes in Computer Science* 14000, Springer, pp. 321–339, doi:10.1007/978-3-031-27481-7_19.
- [5] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*. In: *TACAS (2), Lecture Notes in Computer Science* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.
- [6] Marco Carbone, Luís Cruz-Filipe, Fabrizio Montesi & Agata Murawska (2018): *Multiparty Classical Choreographies*. In: *LOPSTR, Lecture Notes in Computer Science* 11408, Springer, pp. 59–76, doi:10.1007/978-3-030-13838-7_4.
- [7] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *ESOP, Lecture Notes in Computer Science* 4421, Springer, pp. 2–17, doi:10.1007/978-3-540-71316-6_2.

- [8] Marco Carbone, Kohei Honda & Nobuko Yoshida (2012): *Structured Communication-Centered Programming for Web Services*. *ACM Trans. Program. Lang. Syst.* 34(2), pp. 8:1–8:78, doi:10.1145/2220365.2220367.
- [9] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann & Philip Wadler (2016): *Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types*. In: *CONCUR, LIPIcs* 59, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 33:1–33:15, doi:10.4230/LIPIcs.CONCUR.2016.33.
- [10] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In: *POPL*, ACM, pp. 263–274, doi:10.1145/2429069.2429101.
- [11] Marco Carbone, Fabrizio Montesi & Carsten Schürmann (2018): *Choreographies, logically*. *Distributed Comput.* 31(1), pp. 51–67, doi:10.1007/s00446-017-0295-1.
- [12] Marco Carbone, Fabrizio Montesi, Carsten Schürmann & Nobuko Yoshida (2017): *Multiparty session types as coherence proofs*. *Acta Informatica* 54(3), pp. 243–269, doi:10.1007/s00236-016-0285-y.
- [13] Edmund M. Clarke, E. Allen Emerson & A. Prasad Sistla (1986): *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. *ACM Trans. Program. Lang. Syst.* 8(2), pp. 244–263, doi:10.1145/5397.5399.
- [14] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida & Luca Padovani (2016): *Global progress for dynamically interleaved multiparty sessions*. *Mathematical Structures in Computer Science* 26(2), pp. 238–302, doi:10.1017/S0960129514000188.
- [15] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink & Tim A. C. Willemse (2013): *An Overview of the mCRL2 Toolset and Its Recent Advances*. In: *TACAS, Lecture Notes in Computer Science* 7795, Springer, pp. 199–213, doi:10.1007/978-3-642-36742-7_15.
- [16] Luís Cruz-Filipe, Eva Graversen, Lovro Lugovic, Fabrizio Montesi & Marco Peressotti (2022): *Functional Choreographic Programming*. In: *ICTAC, Lecture Notes in Computer Science* 13572, Springer, pp. 212–237, doi:10.1007/978-3-031-17715-6_15.
- [17] Luís Cruz-Filipe, Kim S. Larsen & Fabrizio Montesi (2017): *The Paths to Choreography Extraction*. In: *FoS-SaCS, Lecture Notes in Computer Science* 10203, pp. 424–440, doi:10.1007/978-3-662-54458-7_25.
- [18] Luís Cruz-Filipe & Fabrizio Montesi (2016): *Choreographies in Practice*. In: *FORTE, Lecture Notes in Computer Science* 9688, Springer, pp. 114–123, doi:10.1007/978-3-319-39570-8_8.
- [19] Luís Cruz-Filipe & Fabrizio Montesi (2017): *Encoding asynchrony in choreographies*. In: *SAC*, ACM, pp. 1175–1177, doi:10.1145/3019612.3019901.
- [20] Luís Cruz-Filipe & Fabrizio Montesi (2017): *Procedural Choreographic Programming*. In: *FORTE, Lecture Notes in Computer Science* 10321, Springer, pp. 92–107, doi:10.1007/978-3-319-60225-7_7.
- [21] Luís Cruz-Filipe & Fabrizio Montesi (2020): *A core model for choreographic programming*. *Theor. Comput. Sci.* 802, pp. 38–66, doi:10.1016/j.tcs.2019.07.005.
- [22] Luís Cruz-Filipe, Fabrizio Montesi & Marco Peressotti (2018): *Communications in choreographies, revisited*. In: *SAC*, ACM, pp. 1248–1255, doi:10.1145/3167132.3167267.
- [23] Luís Cruz-Filipe, Fabrizio Montesi & Marco Peressotti (2021): *Certifying Choreography Compilation*. In: *ICTAC, Lecture Notes in Computer Science* 12819, Springer, pp. 115–133, doi:10.1007/978-3-030-85315-0_8.
- [24] Luís Cruz-Filipe, Fabrizio Montesi & Marco Peressotti (2021): *Formalising a Turing-Complete Choreographic Language in Coq*. In: *ITP, LIPIcs* 193, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 15:1–15:18, doi:10.4230/LIPIcs.ITP.2021.15.
- [25] E. Allen Emerson & Edmund M. Clarke (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. *Sci. Comput. Program.* 2(3), pp. 241–266, doi:10.1016/0167-6423(83)90017-5.
- [26] Saverio Giallorenzo, Fabrizio Montesi & Maurizio Gabbriellini (2018): *Applied Choreographies*. In: *FORTE, Lecture Notes in Computer Science* 10854, Springer, pp. 21–40, doi:10.1007/978-3-319-92612-4_2.

- [27] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi & Pascal Weisenburger (2021): *Multiparty Languages: The Choreographic and Multitier Cases (Pearl)*. In: *ECOOP, LIPIcs* 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 22:1–22:27, doi:10.4230/LIPIcs.ECOOP.2021.22.
- [28] Rob J. van Glabbeek & W. P. Weijland (1996): *Branching Time and Abstraction in Bisimulation Semantics*. *J. ACM* 43(3), pp. 555–600, doi:10.1145/233551.233556.
- [29] Thomas T. Hildebrandt, Tijs Slaats, Hugo A. López, Søren Debois & Marco Carbone (2019): *Declarative Choreographies and Liveness*. In: *FORTE, Lecture Notes in Computer Science* 11535, Springer, pp. 129–147, doi:10.1007/978-3-030-21759-4_8.
- [30] Andrew K. Hirsch & Deepak Garg (2022): *Pirouette: higher-order typed functional choreographies*. *Proc. ACM Program. Lang.* 6(POPL), pp. 1–27, doi:10.1145/3498684.
- [31] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP, Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.
- [32] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL, ACM*, pp. 273–284, doi:10.1145/1328438.1328472.
- [33] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.
- [34] Sung-Shik Jongmans & Petra van den Bos (2022): *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*. In: *ESOP, Lecture Notes in Computer Science* 13240, Springer, pp. 520–547, doi:10.1007/978-3-030-99336-8_19.
- [35] Bjørn Angel Kjær, Luís Cruz-Filipe & Fabrizio Montesi (2022): *From Infinity to Choreographies - Extraction for Unbounded Systems*. In: *LOPSTR, Lecture Notes in Computer Science* 13474, Springer, pp. 103–120, doi:10.1007/978-3-031-16767-6_6.
- [36] Wen Kokke, J. Garrett Morris & Philip Wadler (2020): *Towards Races in Linear Logic*. *Log. Methods Comput. Sci.* 16(4), doi:10.23638/LMCS-16(4:15)2020.
- [37] Fabrizio Montesi (2013): *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen. <https://www.fabriziomontesi.com/files/choreographic-programming.pdf>.
- [38] Fabrizio Montesi & Nobuko Yoshida (2013): *Compositional Choreographies*. In: *CONCUR, Lecture Notes in Computer Science* 8052, Springer, pp. 425–439, doi:10.1007/978-3-642-40184-8_30.
- [39] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese & Jacopo Mauro (2015): *Dynamic Choreographies - Safe Runtime Updates of Distributed Applications*. In: *COORDINATION, Lecture Notes in Computer Science* 9037, Springer, pp. 67–82, doi:10.1007/978-3-319-19282-6_5.
- [40] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese & Jacopo Mauro (2017): *Dynamic Choreographies: Theory And Implementation*. *Log. Methods Comput. Sci.* 13(2), doi:10.1007/BF01221097.
- [41] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro & Maurizio Gabbrielli (2014): *AIOGJ: A Choreographic Framework for Safe Adaptive Distributed Applications*. In: *SLE, Lecture Notes in Computer Science* 8706, Springer, pp. 161–170, doi:10.1007/978-3-319-11245-9_9.
- [42] A. Laura Voinea, Ornela Dardha & Simon J. Gay (2019): *Resource Sharing via Capability-Based Multiparty Session Types*. In: *IFM, Lecture Notes in Computer Science* 11918, Springer, pp. 437–455, doi:10.1007/978-3-030-34968-4_24.