Ozone: Fully Out-of-Order Choreographies

Dan Plyukhin
University of Southern Denmark, Denmark

Marco Peressotti
University of Southern Denmark, Denmark

Fabrizio Montesi
University of Southern Denmark, Denmark

Abstract

Choreographic programming is a paradigm for writing distributed applications. It allows programmers to write a single program, called a choreography, that can be compiled to generate correct implementations of each process in the application. Although choreographies provide good static guarantees, they can exhibit high latency when messages or processes are delayed. This is because processes in a choreography typically execute in a fixed, deterministic order, and cannot adapt to the order that messages arrive at runtime. In non-choreographic code, programmers can address this problem by allowing processes to execute out of order—for instance by using futures or reactive programming. However, in choreographic code, out-of-order process execution can lead to serious and subtle bugs, called *communication integrity violations (CIVs)*.

In this paper, we develop a model of choreographic programming for out-of-order processes that guarantees absence of CIVs and deadlocks. As an application of our approach, we also introduce an API for safe non-blocking communication via futures in the choreographic programming language Choral. The API allows processes to execute out of order, participate in multiple choreographies concurrently, and to handle unordered data messages. We provide an illustrative evaluation of our API, showing that out-of-order execution can reduce latency and increase throughput by overlapping communication with computation.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies

Keywords and phrases Choreographic programming, Asynchrony, Concurrency.

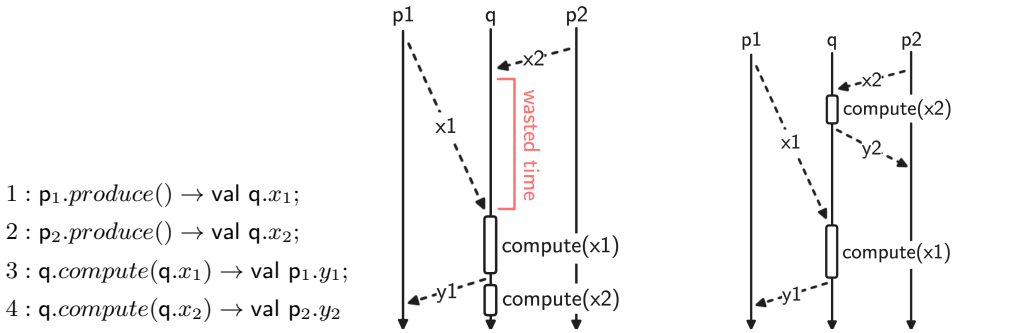
Digital Object Identifier 10.4230/LIPIcs...

Funding Partially supported by Villum Fonden (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Choreographic programming [25] is a paradigm that simplifies writing distributed applications. In contrast to a traditional development style, where one implements a separate program for each type of process in the system, choreographic programming allows a programmer to define the behaviors of all processes together in a single program called a *choreography* [26]. Through *endpoint projection (EPP)*, a choreography can be compiled to generate the programs implementing each process that would otherwise need to be written by hand. Aside from convenience, the advantage of this approach is that certain classes of bugs (such as deadlocks) are impossible *by construction* [8]. Choreographic programming has been applied to popular languages such as Java [16] and Haskell [31], and has been used to implement real-world protocols such as IRC [23].

Processes in choreographic programs typically execute in a fixed, sequential order. Consider Figure 1a, which shows a simple choreography performed by processes p_1, p_2, and q. The syntax $\mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x$ means "p evaluates expression e and sends the result to q,



(a) Choreography **(b)** In-order execution **(c)** Out-of-order execution

Figure 1 A choreography where out-of-order execution can improve performance.

which binds the result to a local variable x". Under the usual semantics for choreographies, $\mathsf{p}_1.produce()$ and $\mathsf{p}_2.produce()$ can be evaluated in parallel because p_1 and p_2 are distinct processes. However, q must execute each step sequentially: first q waits until it receives x_1; then q waits until it receives x_2; and only then can q send p_1 the result of processing x_1.

Figure 1b depicts an execution of the choreography, showing the drawback of a fixed processing order: if x_2 arrives before x_1, q wastes time waiting for x_1 instead of processing x_2. Ideally, q would evaluate $compute(\mathsf{q}.x_1)$ and $compute(\mathsf{q}.x_2)$ according to the arrival order of x_1 and x_2, as shown in Figure 1c. Assuming these two expressions are safe to reorder, such an optimization would allow q to overlap computation with communication and reduce the average latency experienced by p_1 and p_2. We are therefore interested in studying choreographic programming models where processes may execute some statements out of order, or even concurrently. We call such processes *out-of-order processes* and the corresponding choreographies *(fully) out-of-order choreographies*.

Processes with out-of-order features have been considered in prior work. Process models such as the actor model [3] or the π-calculus with delayed receive [24] are expressive enough to implement the behavior in Figure 1c, but these models lack the static guarantees of choreographic programming. More recently, Montesi gave a semantics for *nondeterministic choreographies* [26], i.e., choreographies with nondeterministic choice. Nondeterministic choreographies can implement the execution in Figure 1c, but they are unwieldy when it comes to expressing out-of-order process execution: they require explicitly writing all possible schedulings, lest getting a suboptimal program. For our example, we would get a choreography twice the size of the one in Figure 1a (cf. Section 6). Consequently, nondeterministic choreographies are both hard to write and brittle—a typical drawback when using syntactic operators to express interleavings. This raises the question:

> *Can we develop a choreographic programming model for out-of-order processes that marries the simple syntax of Figure 1a with the semantics of Figure 1c?*

The simplicity of this problem is deceptive, since common-sense approaches can lead to pernicious compiler bugs. For instance, consider Figure 2: two microservices cs (a "content service") and ks (a "key service") send values txt, key to a server s (lines 1 and 2). The server in turn forwards those values to a client c (lines 3 and 4). Notice that if s is an out-of-order process, then it can forward the results in any order, as shown in Figures 2b and 2c. This causes a problem for c: since both txt and key were sent by s, and since both values have the same type (String), c has no way to determine whether the first message contains txt (as

in Figure 2b) or *key* (as in Figure 2c). This problem is easy for compiler writers to miss, leading to disastrous nondeterministic bugs where variables are bound to the wrong values. We call such bugs *communication integrity violations (CIVs)*.

In this paper, we investigate CIVs and other complications that arise from mixing choreographies with out-of-order processes. Our investigation brings forward necessary elements that are missing from previous research on choreographic programming [26] and the neighbouring approach of multiparty session types (which use simpler choreographies without data or computation) [19, 15, 2, 32]. Although the problem in Figure 2 can easily be solved by attaching static information (such as variable names) to each message, we show in Section 2 that a general solution requires mixing static and dynamic information, replicated across multiple processes. We also find that formalizing fully out-of-order choreographies requires several features uncommon in standard choreographic programming models, such as scoped variables and an expanded notion of well-formedness.

We make the following key contributions:

1. We present O_3, a formal model for asynchronous, fully out-of-order choreographies.[1] Our model prevents CIVs by attaching *integrity keys* to messages. A nice consequence of our solution is that messages no longer need to be delivered in FIFO order. We prove that O_3 choreographies ensure deadlock-freedom (Theorem 2) and communication integrity (Theorem 4).
2. We present an EPP algorithm to project O_3 choreographies into out-of-order processes. We prove an operational correspondence theorem, which states that a choreography and its projection evolve in lock-step (Theorem 6). The key to making this proof tractable is a new notion of well-formedness that formalizes a communication integrity invariant. The theorem implies that a correct compiler will not generate code with deadlocks or CIVs.
3. We demonstrate the applicability of our approach by developing *Ozone*, a non-blocking communication API for the choreographic programming language Choral [16].[2] Choreographies implemented with Ozone can use futures [4] to process messages concurrently (as in Figure 1c) without violating communication integrity. We evaluate Ozone with microbenchmarks and a model serving pipeline [33]. Our results confirm that out-of-order execution can dramatically reduce latency and increase throughput for choreographies, putting the performance of hand-written reactive processes within reach of choreographic programmers (we compare to actors written in the popular *Akka* framework [1]).

The paper is structured as follows. Section 2 explores CIVs and other issues in out-of-order choreography models. Section 3 presents our formal model O_3. Section 4 presents our model for out-of-order processes and our EPP. Section 5 presents our non-blocking API for Choral and our evaluation. We conclude with related work in Section 6 and discussion in Section 7.

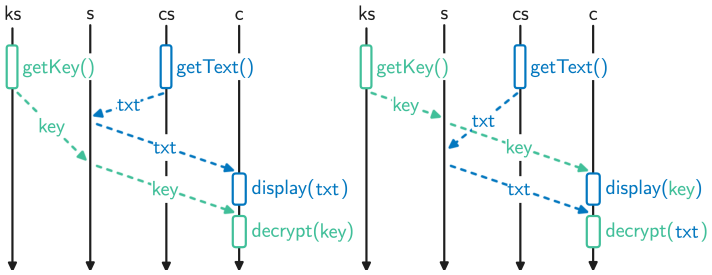
2 Overview

In this section we explore the challenges that must be solved to develop a fully out-of-order choreography model, along with our approach.

[1] The name O_3 derives from our model being *O*ut *O*f *O*rder.
[2] The name *Ozone* derives from O_3 being the chemical formula for ozone.

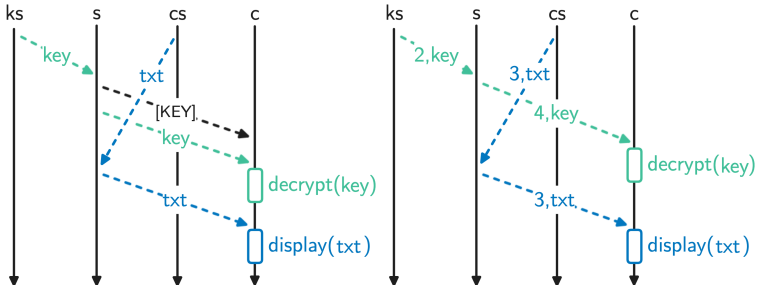
1 : cs.$getText()$ → val s.txt;
2 : ks.$getKey()$ → val s.key;
3 : s.txt → val c.txt;
4 : s.key → val c.key;
5 : c.$display(txt)$;
6 : c.$decrypt(key)$

(a) Choreography



(b) Safe execution **(c)** Buggy execution

Figure 2 A choreography where naïve out-of-order execution is unsafe. Process c cannot distinguish whether the first message it receives represents *key* or *txt*.



(a) Using selections **(b)** Using integrity keys

Figure 3 Two approaches to prevent CIVs: selections and integrity keys.

2.1 Intraprocedural Integrity

Informally, communication integrity is the property that messages communicated in a choreography are bound to the correct variables. To ensure this, processes might need extra information; in Figure 2, process c needs to know which value will arrive first: *txt* or *key*.

A traditional solution would be for s to send a *selection* to c. Selections are communications of constant values, used in choreography languages when one process makes a control flow decision that other processes must follow. Figure 3a shows how s could send the selection [KEY] to inform c that *key* will arrive before *txt*. Indeed, this is the approach used by nondeterministic choreographies [26]. However, selections impose overhead: any time nondeterminism could occur, the programmer would need to insert new selection messages. These extra messages would have both a cognitive cost for the programmer (as programs become littered with selections) and a runtime cost in the form of an extra message.

Instead, we opt to pair each message with a disambiguating tag called an *integrity key*. When c receives a message, it checks the integrity key to find the meaning of the message. Figure 3b uses *line numbers* as integrity keys. For example, the *txt* message is tagged with the number 3 because it was produced by the instruction on line 3 in Figure 2. Equivalently, one could use variable names (assuming that all variables have distinct names), message types (assuming that all messages have distinct types), or operators [7]; essentially these are all ways to combine messages with selections. However, as we will see in the next section, none of these solutions will suffice once we introduce procedures and recursion.

Integrity keys have another advantage over selections: they make it safe for the network to reorder messages. For instance, the selection in Figure 3a will only prevent CIVs if *key* and *txt* are delivered in the same order they were sent. Thus previous theories and implementations

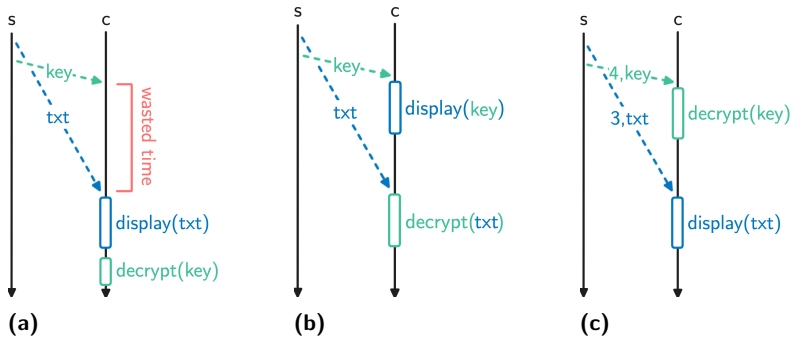


Figure 4 The challenges of non-FIFO delivery. Part (a) depicts head-of-line blocking when using a FIFO transport protocol: The message containing k arrives first, but it cannot be processed until t arrives. Part (b) depicts a CIV caused by using an unordered transport protocol without integrity keys. Part (c) depicts how the processes can use integrity keys to prevent CIVs.

of choreographic languages require a transport protocol that ensures reliable FIFO communication [16, 26]. These models are therefore susceptible to head-of-line blocking [30], where one delayed message can prevent others from being processed (Figure 4a). Figure 4b shows why FIFO is necessary in these models: unordered messages can cause CIVs. Because our model combines unordered messages, integrity keys, and out-of-order processes, it circumvents the head-of-line blocking problem—as shown in Figure 4c.

2.2 Procedural Choreographies

Choreographies can use procedures parameterised on processes for modularity and recursion [11, 26]. Figure 5a shows an example: a procedure X with three *roles* (i.e., process parameters) $\mathsf{a}, \mathsf{b}, \mathsf{c}$. The procedure X is invoked twice—once with processes $\mathsf{p}, \mathsf{q}, \mathsf{r}_1$ (line 7) and again with $\mathsf{p}, \mathsf{q}, \mathsf{r}_2$ (line 8). In the body of X, role a produces a value and sends it to b; then b transforms the value and sends it to c; finally, c processes the value and sends it to a. As usual in most programming languages, we will assume the variables $\mathsf{a}.w, \mathsf{b}.x, \mathsf{c}.y$, and $\mathsf{a}.z$ are locally scoped—this is in contrast to many choreography models, where variables at processes are all mutable fields accessible anywhere in the program.

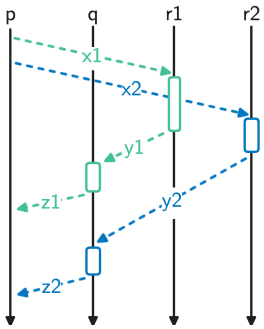
In existing choreography models, a process can only participate in one choreographic procedure at a time. This is no longer the case with fully out-of-order choreographies. Consider Figure 5a, where process p invokes procedure X twice. The process may begin by invoking the first procedure call (line 7), computing $\mathsf{p}.w$ (line 2), and sending $\mathsf{p}.w$ to r_1 (line 3). Then, instead of executing its next instruction—i.e. becoming blocked by waiting for a message on line 5—p can skip the instruction and proceed to invoke the second procedure call (line 8). Thus, we can have an execution like in Figure 5b, in which p sends a message to r_1 as part of the first procedure call and immediately sends a message to r_2 as part of the second procedure call. This unusual semantics is exactly what we would expect in a choreography language with non-blocking receive—such as in Choral, when using the Ozone API to bind the result of a communication to a future (Section 5).

2.2.1 Interprocedural Integrity

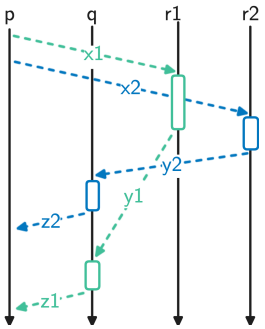
Concurrent choreographic procedures add another dimension of complexity to the communication integrity problem. Figures 5b and 5c show why: depending on the order that r_1 and

1 : $X(\mathsf{a}, \mathsf{b}, \mathsf{c}) =$
2 : $\quad \mathsf{val}\ \mathsf{a}.w = produce();$
3 : $\quad \mathsf{a}.w \to \mathsf{val}\ \mathsf{b}.x;$
4 : $\quad \mathsf{b}.transform(x) \to \mathsf{val}\ \mathsf{c}.y;$
5 : $\quad \mathsf{c}.process(y) \to \mathsf{val}\ \mathsf{a}.z;$
6 : $\quad \mathsf{a}.store(w, z)$
7 : $X(\mathsf{p}, \mathsf{r}_1, \mathsf{q});$
8 : $X(\mathsf{p}, \mathsf{r}_2, \mathsf{q})$

(a)



(b)



(c)

Figure 5 A choreography and two possible executions. In both diagrams, the green lines correspond to $X(\mathsf{p}, \mathsf{q}, \mathsf{r}_1)$ and the blue lines correspond to $X(\mathsf{p}, \mathsf{q}, \mathsf{r}_2)$.

r_2's messages arrive at q, the messages from q may arrive at p in any order. (This occurs even if we assume reliable FIFO delivery!) Like in the previous section, p cannot distinguish which message pertains to which procedure invocation. But now static information is insufficient to ensure communication integrity: both messages from q pertain to the same variable in the same procedure, so the integrity keys fail to distinguish the different procedure calls. We call this the *interprocedural CIV problem*.

The example above shows that integrity keys need dynamic information prevent CIVs. We can solve the problem by combining the line numbers used in Section 2.1 with some *session token* t that uniquely identifies each procedure invocation. Applied to Figures 5b and 5c, p could inspect the session token to determine whether the messages pertain to the first procedure call (line 7) or the second (line 8). But this requires p and q to somehow achieve *a priori* agreement about which tokens correspond to which procedure invocations.

One solution to the interprocedural CIV problem would be to select a "leader" process for each procedure call, and let the leader compute a session token for all the other roles to use. However, this would make the leader a bottleneck: until the other participants receive the token, senders would not be able to send messages, and recipients would not be able to discern which procedure invocation their incoming messages pertain to. We therefore propose a method for processes to compute session tokens independently, using only local data, such that they still agree on the same value of the token for each procedure invocation.

Observe that a procedure call is uniquely identified by its caller (i.e. the procedure call that called it) and its line number l. Assuming the caller already has a unique token t, the callee's token can be computed as some injective function $\mathsf{nextToken}(l, t)$. This function would need to satisfy two properties:

- **Determinism:** For any input pair l, t, $\mathsf{nextToken}(l, t)$ always produces the same value t'.
- **Injectivity:** Distinct input pairs l, t produce distinct output tokens.

Determinism ensures that if two processes in the same procedure call (with token t) invoke the same procedure (on line l) then both processes will agree on the value of $\mathsf{nextToken}(l, t)$. *Injectivity* ensures that if a process concurrently participates in two different procedure calls (with distinct tokens t_1, t_2) and invokes two procedures (on lines l_1, l_2—possibly $l_1 = l_2$) then the resulting session tokens will be distinct ($\mathsf{nextToken}(l_1, t_1) \neq \mathsf{nextToken}(l_2, t_2)$). In the next section, we realize these constraints by representing tokens as lists of line numbers and defining $\mathsf{nextToken}$ to be the list-prepend operator.

$$
\begin{array}{lll|lll}
\mathscr{C} ::= \{X_i(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C_i\}_{i\in\mathcal{I}} & & \textit{(decls)} & & & \\
C ::= I;\, C & & \textit{(seq)} & \mid \{\, C \,\} & & \textit{(block)} \\
\mid 0 & & \textit{(end)} & & & \\
I ::= l, t : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x & & \textit{(comm)} & \mid l, t : \mathsf{p} \to \mathsf{q}[L] & & \textit{(sel)} \\
\mid l, t : \mathsf{val}\ \mathsf{p}.x = e & & \textit{(expr)} & \mid l, t : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 & & \textit{(cond)} \\
\mid l, t : X(\overline{\mathsf{p}}, \overline{a}) & & \textit{(call)} & \mid l, t : \mathsf{p} \rightsquigarrow \mathsf{q}.x & & \textit{(comm}^\dagger\textit{)} \\
\mid l, t : \mathsf{p} \rightsquigarrow \mathsf{q}[L] & & \textit{(sel}^\dagger\textit{)} & \mid l, t : \overline{\mathsf{p}}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C \,\} & & \textit{(call}^\dagger\textit{)} \\
t ::= \mathsf{t} & & \textit{(placeholder)} & \mid \tau & & \textit{(token}^\dagger\textit{)} \\
e ::= f(\overline{e}) & & \textit{(app)} & \mid a & & \textit{(atom)} \\
a ::= v@\mathsf{p} & & \textit{(val)} & \mid \mathsf{p}.x & & \textit{(var)}
\end{array}
$$

Figure 6 Syntax for choreographies in O_3. Terms marked with † only appear at runtime.

3 Choreography Model

In this section we present O_3, a formal model for asynchronous, fully out-of-order choreographies. Statements can be executed in any order (up to data dependency) and messages can be delivered out of order. The section concludes with proofs of deadlock-freedom and communication integrity.

3.1 Syntax

The syntax for choreographies in O_3 is defined by the grammar in Figure 6. Two example choreographies are shown in Figure 7; we explain their semantics in Section 3.2.2.

A choreography C is executed in the context of a collection of *procedures* $\mathscr{C}$. Each procedure $X_i(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C_i$ is parameterized by a list of *roles* $\overline{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n$ and role-local *parameters* $\overline{\mathsf{p}.x} = \mathsf{p}_{j_1}.x_1, \ldots, \mathsf{p}_{j_m}.x_m$ where every parameter $\mathsf{p}_{j_k}.x_k$ is located at one of the roles in $\overline{\mathsf{p}}$. We assume that procedures do not contain runtime terms (such as $l, t : \mathsf{p} \rightsquigarrow \mathsf{q}.x$).

A choreography C consists of a sequence of *instructions* I, followed by the end symbol 0 which is often omitted. Each instruction is prefixed with a *line number* l and a *token* t. We call this pair an *integrity key*. If C_i is the body of a procedure in $\mathscr{C}$, then the token t on every instruction in C_i must be a *token placeholder* t. When the procedure is invoked, all token placeholders t in C_i will be replaced with a fresh *token value* τ.

We assume that line numbers in $\mathscr{C}$ are similar to line numbers in a real computer program: Each instruction I in $\mathscr{C}$ has a distinct line number l. When a procedure $X_i(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C_i$ is invoked, the line numbers in C_i will remain unchanged. This will allow us to access the static location of an instruction at runtime in order to compute the integrity key.

There are five kinds of instructions. A communication $\mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x;\, C$ instructs process p to evaluate expression e and send it to process q, which will bind the result to $\mathsf{q}.x$ in the continuation C. A selection $\mathsf{p} \to \mathsf{q}[L];\, C$ conveys knowledge of choice [26]: it instructs p to send a value literal L to q, informing q that a decision (represented by L) has been made. A local computation $\mathsf{val}\ \mathsf{p}.x = e;\, C$ instructs p to evaluate e and bind the result to $\mathsf{p}.x$ in C. A conditional $\mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;\, C$ instructs p to evaluate e and for the processes to proceed with C_1 or C_2 according to the result. A procedure call $X_i(\overline{\mathsf{p}}, \overline{a});\, C$ instructs processes $\overline{\mathsf{p}}$ to invoke procedure $X_i(\overline{\mathsf{q}}, \overline{\mathsf{q}.y}) = C_i$ defined in $\mathscr{C}$, with processes $\overline{\mathsf{p}}$ playing roles

$$
\begin{aligned}
&\mathit{BuyItem}(\mathsf{s},\ \mathsf{b},\ \mathsf{b}.\mathit{itemID}) = \\
&\quad 1, \mathsf{t} : \mathsf{b}.\mathit{itemID} \to \mathsf{val}\ \mathsf{s}.\mathit{itemID}; \\
&\quad 2, \mathsf{t} : \mathsf{val}\ \mathsf{s}.\mathit{item?} = \mathit{sell}(\mathsf{s}.\mathit{itemID}); \\
&\quad 3, \mathsf{t} : \mathsf{s}.\mathit{item?} \to \mathsf{val}\ \mathsf{b}.\mathit{item?} \\
&4, \tau_0 : \mathit{BuyItem}(\mathsf{seller}, \mathsf{buyer}_1, 123@\mathsf{buyer}_1); \\
&5, \tau_0 : \mathit{BuyItem}(\mathsf{seller}, \mathsf{buyer}_2, 543@\mathsf{buyer}_2)
\end{aligned}
$$

(a)

$$
\begin{aligned}
&\mathit{StreamIt}(\mathsf{p},\ \mathsf{c}) = \\
&\quad 1, \mathsf{t} : \mathsf{p}.\mathit{produce}() \to \mathsf{val}\ \mathsf{c}.x; \\
&\quad 2, \mathsf{t} : \mathsf{val}\ \mathsf{c}.z = \mathit{consume}(\mathsf{c}.x); \\
&\quad 3, \mathsf{t} : \mathsf{if}\ (\mathit{itemsLeft}() > 0)@\mathsf{p}\ \mathsf{then} \\
&\qquad 4, \mathsf{t} : \mathsf{p} \to \mathsf{c}[\textsc{More}] \\
&\qquad 5, \mathsf{t} : \mathit{StreamIt}(\mathsf{p}, \mathsf{c}) \\
&\quad \mathsf{else} \\
&\qquad 6, \mathsf{t} : \mathsf{p} \to \mathsf{c}[\textsc{Done}] \\
&7, \tau_0 : \mathit{StreamIt}(\mathsf{p}_1, \mathsf{c}); \\
&8, \tau_0 : \mathit{StreamIt}(\mathsf{p}_2, \mathsf{c})
\end{aligned}
$$

(b)

Figure 7 Two example choreographies. On the left, processes buyer_1 and buyer_2 concurrently attempt to buy products from seller. On the right, producers p_1 and p_2 concurrently send streams of data to a shared consumer c.

$\overline{\mathsf{q}}$ and arguments $\overline{a}$ (which may take the form of values $v@\mathsf{p}$ or variables $\mathsf{p}.x$) substituted for parameters $\overline{\mathsf{q}.y}$ in C_i. In addition to these basic instructions, a choreography may contain blocks $\{\, C \,\}; C'$ which limit the scope of variables defined in C so they do not extend to C'.

In addition, choreographies can contain *runtime instructions* that represent an instruction in progress; these terms are an artifact of the semantics, not written explicitly by the programmer. A communication-in-progress $\mathsf{p} \rightsquigarrow \mathsf{q}.x$ indicates that p sent a message to q, which q has not yet received. Similarly, a selection-in-progress $\mathsf{p} \rightsquigarrow \mathsf{q}[L]$ indicates that p sent a selection. A procedure-call-in-progress $\overline{\mathsf{p}}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C \,\}$ indicates that some processes have invoked X, and others have not—we leave the details to Section 3.2.

Expressions e are composed of *atoms* a (i.e. variables $\mathsf{p}.x$ and values $v@p$) and function applications $f(\overline{e})$. Although the variables $\mathsf{p}.x$ are immutable, we assume that a function f evaluated by p can mutate p's state as a side-effect. Technically, having side-effects in our theory is not necessary. However, most choreographic programming theories and implementations equip processes with mutable state [26]; this includes Choral, the language we use to implement the Ozone API in Section 5.

3.2 Semantics

We now give a fully out-of-order semantics for choreographies in O_3. The semantics is a labelled transition system on *configurations* $\langle\, C,\ \Sigma,\ K \,\rangle$, where C is a choreography, Σ is a mapping from process names p to process states σ, and K is a mapping from process names p to multisets of messages M yet to be delivered to p. We also assume there exists a set of unchanging procedure declarations $\mathscr{C}$, not shown explicitly in the configuration.

An *initial configuration* is a configuration $\langle\, C,\ \Sigma,\ K \,\rangle$ where Σ maps each p to an arbitrary state, K maps each p to the empty set, and all instructions in C use the same token τ_0, called the *initial token*. We assume initial configurations to be well-formed, cf. Section 3.3. The transition relation $(\xrightarrow{\mathsf{p}})$ is on configurations, where p identifies which process took a step.

Messages in our semantics are represented as triples (l, τ, v). Here l is the line number of the communication that sent the message, τ is the token associated with the procedure

invocation that sent the message, and v is a value called the *payload*. Together, the pair (l, τ) is called the *integrity key* of the message; the line number prevents intraprocedural CIVs (Section 2.1) while the token prevents interprocedural CIVs (Section 2.2).

3.2.1 Transition rules

Figure 8 defines the semantics for O_3, which extends textbook models for procedural and asynchronous choreographies to allow full out-of-order execution [26]. That is, in a choreography of the form $I_1; I_2; C$, the statement I_2 can always be executed before I_1 *unless*:

1. *(Data dependency)* I_1 binds a variable $\mathsf{p}.x$ that is used in I_2; or
2. *(Control dependency)* I_1 is a selection of the form $\mathsf{p} \to \mathsf{q}[L]$ or $\mathsf{p} \rightsquigarrow \mathsf{q}[L]$, and I_2 is an action performed by q.

The semantics for communication is defined by rules C-Send and C-Recv. In C-Send for the communication term $l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x$, the expression e is evaluated in the context of p's state using the notation $\Sigma(\mathsf{p}) \vdash e \Downarrow (v, \sigma)$. Evaluating e produces a value v and a new state σ for p; we assume that $(\vdash)$ is defined for any e that contains no free variables and for any state $\Sigma(\mathsf{p})$. The C-Send rule transforms the communication term into a communication-in-progress term $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ and adds the message (l, τ, v) to q's set of undelivered messages. The message can subsequently be received by q using the C-Recv rule. This rule removes the communication-in-progress term and substitutes the message payload v into the continuation C. Notice that the integrity key l, τ of the message is matched against the integrity key of the communication-in-progress, $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$. Notice also that the semantics for communication is not defined if the token t is merely a placeholder t—it must be a token *value* τ. Indeed, in Section 3.3 we show that placeholders only appear in $\mathscr{C}$, never in C.

Rules C-Select and C-OnSelect closely mirror the semantics of C-Send and C-Recv—the key difference is that a label L is communicated instead of a value. Rules C-Compute and C-If are standard, except for changes made to use lexical scope instead of global scope: C-Compute substitutes the value v into the continuation C (instead of storing it in the local state Σ) and C-If places the continuation C_i in a block to prevent variable capture. To garbage collect empty blocks, C-If uses a concatenation operator $(\fatsemi)$ defined as:

$$\{I; C\} \fatsemi C' = \{I; C\};\ C' \qquad\qquad \{0\} \fatsemi C' = C'$$

The C-Delay rule is used in choreography models to enable a limited form of out-of-order execution, where unrelated processes execute concurrently: given a choreography $I; C$, C-Delay would ordinarily prevent any q from executing in C if q is somehow involved in I. Our formulation of the rule is weakened: q is only prevented from executing in C if I is a selection at q, i.e. a control dependency. The rule still respects data dependencies, however, by design of the other rules—for instance, $l, \tau : \mathsf{p}.x \to \mathsf{val}\ \mathsf{q}.y$ cannot be evaluated until x is bound to a value. Thus our version of C-Delay enables *full* out-of-order execution.

The rules C-First, C-Enter, C-Last, and C-Delay-Proc model procedure calls, with extra machinery to model how processes can execute their roles in a choreographic procedure in parallel until they need to interact. Given a procedure call $l, \tau : X(\overline{\mathsf{p}}, \overline{a})$, C-First models how $\mathsf{p} \in \overline{\mathsf{p}}$ has entered the procedure before any of the other processes. The rule replaces the procedure call with a procedure-call-in-progress $l, \tau : \overline{\mathsf{p}} \setminus \mathsf{p}.\ X(\overline{\mathsf{p}}, \overline{a})\ \{\ C_1'\ \}$ to reflect this fact; the choreography C_1' is the body of the procedure, which p may begin executing via the C-Delay-Proc rule. The remaining processes can enter the procedure via the C-Enter rule, and the last process to enter the procedure uses the C-Last rule. As we explain below, these rules also compute new integrity keys for the callee procedure to prevent CIVs.

The key novelty of our semantics for procedures is the use of nextToken. In C-First, the body C_1' is obtained by computing the token $\tau' = \mathsf{nextToken}(l, \tau)$ and substituting τ' for all occurrences of the token placeholder t. Notice that the semantics makes it appear as if the processes have synchronized to compute the next token; in Section 4, we give a semantics where each process computes the next token independently and in Theorem 6 we prove that the two models correspond. Hence the apparent synchronization has no runtime cost.

As discussed in Section 2.2.1, $\mathsf{nextToken} : \mathbb{N} \times \mathsf{Token} \to \mathsf{Token}$ is a pure injective function for computing new tokens (of type Token) using integrity keys (of type $\mathbb{N} \times \mathsf{Token}$). To ensure the integrity keys from two concurrent procedures never collide, nextToken must produce unique, non-repeating keys upon iterated application. One way this can be realized is by representing $\mathsf{Token} = \mathbb{N}^*$ as lists of numbers, the initial token τ_0 as an empty list $[]$, and implementing $\mathsf{nextToken}(l, \tau) = l :: \tau$, i.e. prepending the line number l to the list. Intuitively, this means the token associated with a procedure invocation is a simplified *call stack* of line numbers from which the procedure was called.

3.2.2 Discussion

Figure 7a expresses a choreography in which two buyer processes concurrently buy items from a seller process. In the initial configuration, buyer_1 can enter the procedure on line 4, buyer_2 can enter the procedure on line 5, and seller can enter either procedure. If buyer_2 enters first (using C-Delay and C-Enter), it can proceed to send $543@\mathsf{buyer}_2$ to seller (using C-Com). Then seller can enter the procedure on line 5 (using C-Delay and C-Last) and proceed to receive the message from buyer_2 (using C-Recv). This execution would be impossible in a standard choreography model because seller would need to complete the procedure invocation on line 4 before it could enter the procedure on line 5. The added concurrency ensures that slowness in buyer_1 does not prevent buyer_2 from making progress.

Notice the out-of-order semantics of Figure 7a also adds nondeterminism. Suppose buyer_1 and buyer_2 attempt to buy the same item and the seller only has one copy. One of the buyers will receive the item, and the other will receive a null value. In a standard choreography model, the item would always go to buyer_1. In O_3, the item will be sold nondeterministically according to the order that messages arrive to the seller. This nondeterminism can be problematic—it makes reasoning about choreographies harder—but also increases expressivity: nondeterminism is essential in distributed algorithms like consensus and leader election. Reasoning about nondeterminism in choreographies is an important topic for future work.

Figure 7b shows we can also express recursive choreographies. In each iteration of the procedure *StreamIt*, a producer p sends a value to a consumer c (line 1) and decides whether to start another iteration (line 3). Then the producer asynchronously informs the consumer about its decision (lines 4 and 6) and can proceed with the next iteration (line 5) without waiting for the consumer. Because messages in O_3 are unordered, the consumer can consume items (line 2) from different iterations in any order; this prevents head-of-line blocking [30].

In the initial choreography of Figure 7b, producers $\mathsf{p}_1, \mathsf{p}_2$ and a consumer c invoke two instances of *StreamIt*. As in Figure 7a, the two procedures evolve concurrently; a slowdown in p_1 will not prevent c from consuming items produced by p_2.

3.3 Properties

In this section we prove that O_3 choreographies are deadlock-free and we formalize the communication integrity property. Combined with the EPP Theorem presented in Section 4, these results imply that projected code inherits the same properties.

$$\frac{\Sigma(\mathsf{p}) \vdash e \Downarrow (v, \sigma) \qquad M = K(\mathsf{q}) \uplus \{(l, \tau, v)\}}{\left\langle\, l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x;\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x;\ C,\ \Sigma[\mathsf{p} \mapsto \sigma],\ K[\mathsf{q} \mapsto M] \,\right\rangle}\ \text{C-Send}$$

$$\frac{(l, \tau, v) \in K(\mathsf{q}) \qquad M = K(\mathsf{q}) \setminus \{(l, \tau, v)\}}{\left\langle\, l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x;\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{q}} \left\langle\, C[\mathsf{q}.x \mapsto v@q],\ \Sigma,\ K[\mathsf{q} \mapsto M] \,\right\rangle}\ \text{C-Recv}$$

$$\frac{M = K(\mathsf{q}) \cup \{(l, \tau, L)\}}{\left\langle\, l, \tau : \mathsf{p} \to \mathsf{q}[L];\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}[L];\ C,\ \Sigma,\ K[\mathsf{q} \mapsto M] \,\right\rangle}\ \text{C-Select}$$

$$\frac{K(\mathsf{q}) = \{(l, \tau, L)\} \cup M}{\left\langle\, l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}[L];\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{q}} \left\langle\, C,\ \Sigma,\ K[\mathsf{q} \mapsto M] \,\right\rangle}\ \text{C-OnSelect}$$

$$\frac{\Sigma(\mathsf{p}) \vdash e \Downarrow (v, \sigma)}{\left\langle\, l, \tau : \mathsf{val}\ \mathsf{p}.x = e;\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, C[\mathsf{p}.x \mapsto v@\mathsf{p}],\ \Sigma[\mathsf{p} \mapsto \sigma],\ K \,\right\rangle}\ \text{C-Compute}$$

$$\frac{\Sigma(\mathsf{p}) \vdash e \Downarrow v \qquad \text{if } v = \mathsf{true} \text{ then } i = 1 \text{ else } i = 2}{\left\langle\, l, \tau : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, \{\, C_i \,\} \fatsemi C,\ \Sigma,\ K \,\right\rangle}\ \text{C-If}$$

$$\frac{\left\langle\, C_1,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, C_1',\ \Sigma',\ K' \,\right\rangle}{\left\langle\, \{\, C_1 \,\};\ C_2,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, \{\, C_1' \,\} \fatsemi C_2,\ \Sigma',\ K' \,\right\rangle}\ \text{C-Block}$$

$$\frac{\left\langle\, C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{q}} \left\langle\, C',\ \Sigma',\ K' \,\right\rangle \qquad I \text{ is not a selection at } \mathsf{q}}{\left\langle\, I;\ C,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{q}} \left\langle\, I;\ C',\ \Sigma',\ K' \,\right\rangle}\ \text{C-Delay}$$

$$\frac{\begin{array}{c}(X(\overline{\mathsf{q}}, \overline{\mathsf{q}.y}) = C_1) \in \mathscr{C} \qquad C_1' = C_1[\overline{\mathsf{q}}, \overline{\mathsf{q}.y}, \mathsf{t} \mapsto \overline{\mathsf{p}}, \overline{a}, \tau'] \\ \mathsf{p} \in \overline{\mathsf{p}} \qquad \tau' = \mathsf{nextToken}(l, \tau)\end{array}}{\left\langle\, l, \tau : X(\overline{\mathsf{p}}, \overline{a});\ C_2,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, l, \tau : \overline{\mathsf{p}} \setminus \mathsf{p}.\, X(\overline{\mathsf{p}}, \overline{a})\, \{\, C_1' \,\};\ C_2,\ \Sigma,\ K \,\right\rangle}\ \text{C-First}$$

$$\frac{\mathsf{p} \in \overline{\mathsf{p}}}{\left\langle\, l, \tau : \overline{\mathsf{p}}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C_1 \,\};\ C_2,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, l, \tau : \overline{\mathsf{p}} \setminus \mathsf{p}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C_1 \,\};\ C_2,\ \Sigma,\ K \,\right\rangle}\ \text{C-Enter}$$

$$\frac{}{\left\langle\, l, \tau : \mathsf{p}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C_1 \,\};\ C_2,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, \{\, C_1 \,\} \fatsemi C_2,\ \Sigma,\ K \,\right\rangle}\ \text{C-Last}$$

$$\frac{\left\langle\, C_1,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, C_1',\ \Sigma',\ K' \,\right\rangle \qquad \mathsf{p} \notin \overline{\mathsf{p}}}{\left\langle\, l, \tau : \overline{\mathsf{p}}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C_1 \,\};\ C_2,\ \Sigma,\ K \,\right\rangle \xrightarrow{\mathsf{p}} \left\langle\, l, \tau : \overline{\mathsf{p}}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C_1' \,\};\ C_2,\ \Sigma',\ K' \,\right\rangle}\ \text{C-Delay-Proc}$$

Figure 8 Semantics for fully out-of-order choreographies

To prove these properties we need an invariant that characterizes how the rules of O_3 preserve the intuition from Section 2. For example, consider the following configurations:

$$\langle\, l, \tau_0 : \mathsf{p} \rightsquigarrow \mathsf{q}.x,\ \Sigma,\ \{\mathsf{p} \mapsto \emptyset, \mathsf{q} \mapsto \emptyset\} \,\rangle \tag{1}$$

$$\langle\, l, \tau_0 : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x,\ \Sigma,\ \{\mathsf{p} \mapsto \emptyset, \mathsf{q} \mapsto \{(l, \tau, v)\}\} \,\rangle \tag{2}$$

$$\langle\, \{1, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x\}; \{1, \tau : \mathsf{p}.e' \to \mathsf{val}\ \mathsf{q}.x\},\ \Sigma,\ \{\mathsf{p} \mapsto \emptyset, \mathsf{q} \mapsto \emptyset\} \,\rangle \tag{3}$$

$$\langle\, 3, \tau_0 : \mathsf{p}.\, X(\mathsf{p}, \mathsf{q})\, \{\, 1, \tau_0 : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x \,\},\ \Sigma,\ \{\mathsf{p} \mapsto \emptyset, \mathsf{q} \mapsto \emptyset\} \,\rangle \tag{4}$$

Configuration (1) is not reachable because $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ never occurs unless q has an undelivered message from p. Dually, configuration (2) is not reachable because p has a message in its queue that, according to the choreography, has not yet been sent. Configuration (3) is unreachable because the two instructions share the same integrity key; we will show that $\mathsf{nextToken}$ ensures such configurations never arise. Likewise, $\mathsf{nextToken}$ also forbids configuration (4), since the token of the instruction $1, \tau_0 : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x$ must have been derived from the integrity key of the enclosing call $3, \tau_0 : \mathsf{p}.\, X(\mathsf{p}, \mathsf{q})\, \{\ldots\}$. Specifically, $\tau_0 \neq \mathsf{nextToken}(3, \tau_0)$. To specify this last property, recall that tokens are represented as lists of integers $l_1 :: l_2 :: \ldots$. We say (l_1, t_1) is a *prefix* of (l_2, t_2)—written $(l_1, t_1) \prec (l_2, t_2)$—if the list $l_1 :: t_1$ is a prefix of $l_2 :: t_2$ and that the keys are *disjoint* if neither is a prefix of the other.

Following convention, we formalize the properties of reachable configurations by defining which configurations and procedures are *well-formed*. Figure 9 highlights the most interesting rules that define well-formedness, where ✓ reads 'well-formed' – the rest can be found in Appendix A. In particular, well-formedness ensures that:

1. (C-WF-Send) $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ occurs in C if and only if $(l, \tau, v) \in K(\mathsf{q})$ for some v.
2. (C-WF-Select) $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}[L]$ occurs in C if and only if $(l, \tau, L) \in K(\mathsf{q})$.
3. (C-WF-Def) Each I in C has a distinct integrity key l, t, where t is not a placeholder.
4. (C-WF-Calling) If the integrity key of I is a prefix of the integrity key of I' then I is a communication-in-progress $l, t : \overline{\mathsf{p}}.\, X(\overline{\mathsf{p}}, \overline{a})\, \{\, C' \,\}$ and I' is in C'.

Well-formedness also guarantees other properties seen in other choreography models, e.g., that procedures contain no free variables and that processes waiting to enter a procedure have the same local behaviour in the original procedure body and the current choreography [26]. As in prior work [26, 13], the latter check is made by using endpoint projection ($[\![C]\!]_{\mathsf{p}}$), which returns the local behaviour of a process in a choreography and is defined later in Section 4.3.

▶ **Theorem 1** (Preservation). *If* $\langle\, C,\ \Sigma,\ K \,\rangle$ *is well-formed and* $\langle\, C,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, C',\ \Sigma',\ K' \,\rangle$, *then* $\langle\, C',\ \Sigma',\ K' \,\rangle$ *is well-formed.*

Proof. By induction on the rules of $\xrightarrow{\mathsf{p}}$. We focus on the rules for communication and procedure invocation.

C-Send replaces $l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x$ with $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ and adds a message (l, τ, v). By the induction hypothesis, (l, τ, v) is not already in K.

C-Recv eliminates $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ and removes a message (l, τ, v). Since each instruction has a distinct integrity key by hypothesis, no other $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ term occurs in C.

C-First introduces new terms into the choreography by invoking the call $l_1, \tau_1 : X(\overline{\mathsf{p}}, \overline{a})$. By the induction hypothesis, for any other instruction $l_2, t_2 : I$ in C, either (a) keys l_1, t_1 and l_2, t_2 are disjoint; or (b) $l_2, t_2 : I$ is a call-in-progress containing $l_1, \tau_1 : X(\overline{\mathsf{p}}, \overline{a})$. In case (a), disjointness implies any instruction in the body of the procedure $C'[\overline{\mathsf{q}}, \overline{\mathsf{q}.y}, \mathsf{t} \mapsto \overline{\mathsf{p}}, \overline{\mathsf{p}.x}, \tau']$ will also have a key that is disjoint from l_2, t_2. In case (b), notice $\forall l,\ (l_2, t_2) \prec (l_1, t_1) \prec (l, \mathsf{nextToken}(l_1, t_1))$; hence any interaction in the body has a key of which (l_2, t_2) is a prefix. ◀

$$\frac{\forall v,\ (l,\tau,v) \notin K(\mathsf{q})}{\langle l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x,\ K\rangle \checkmark}\ \text{C-WF-Send} \qquad \frac{(l,\tau,L) \notin K(\mathsf{q})}{\langle l, \tau : \mathsf{p} \to \mathsf{q}[L],\ K\rangle \checkmark}\ \text{C-WF-Select}$$

$$\frac{\begin{array}{c}\overline{\mathsf{p}}\ \text{distinct} \qquad \overline{\mathsf{p}.x}\ \text{distinct} \qquad \mathsf{pn}(C) \subseteq \overline{\mathsf{p}} \\ \forall \mathsf{p}.x \in \overline{\mathsf{p}.x},\ \mathsf{p} \in \overline{\mathsf{p}} \qquad \langle I, K\rangle \checkmark \text{ for each } I \in \mathsf{stats}(C) \\ C \text{ contains no runtime terms} \qquad \mathsf{keys}(C)\ \text{distinct} \qquad \forall (l,t) \in \mathsf{keys}(C),\ t = \mathsf{t}\end{array}}{X(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C \checkmark}\ \text{C-WF-Def}$$

$$\frac{\begin{array}{c}\langle l, \tau : X(\overline{\mathsf{q}}, \overline{a}),\ K\rangle \checkmark \qquad (X(\mathsf{q}_1, \ldots, \mathsf{q}_n, \mathsf{q}^1.x_1, \ldots, q^m.x_m) = C') \in \mathscr{C} \\ \{\mathsf{r}_1, \ldots, \mathsf{r}_k\} \subseteq \{\mathsf{p}_1, \ldots, \mathsf{p}_n\} \qquad \forall i \le k, j \le n \ \text{ if } \mathsf{r}_i = \mathsf{p}_j \text{ then } [\![C]\!]_{\mathsf{r}_i} = [\![C']\!]_{\mathsf{q}_j}\end{array}}{\langle l, \tau : \mathsf{r}_1, \ldots, \mathsf{r}_k.\, X(\mathsf{p}_1, \ldots, \mathsf{p}_n, a_1, \ldots, a_m)\,\{\, C \,\},\ K\rangle \checkmark}\ \text{C-WF-Calling}$$

$\mathsf{stats}(0) = \epsilon$
$\mathsf{stats}(I;\ C) = \mathsf{stats}(I),\ \mathsf{stats}(C)$
$\mathsf{stats}(\{\, C \,\}) = \mathsf{stats}(C)$
$\mathsf{stats}(l, t : \mathsf{if}\ e@\mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) =$
$\quad (l, t : \mathsf{if}\ e@\mathsf{q}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2),\ \mathsf{stats}(C_1),\ \mathsf{stats}(C_2)$
$\mathsf{stats}(l, t : \overline{\mathsf{q}}.\, X(\overline{\mathsf{p}}, \overline{a})\,\{\, C \,\}) =$
$\quad (l, t : \overline{\mathsf{q}}.\, X(\overline{\mathsf{p}}, \overline{a})\,\{\, C \,\}), \mathsf{stats}(C)$
$\mathsf{stats}(l, t : \eta) = (l, t : \eta)$ otherwise
$\mathsf{stats}(C) = [\mathsf{stats}(C) \mid \mathsf{p} \in \mathsf{pn}(C)]$
$\mathsf{keys}(C) = [(l, t) \mid (l, t : \eta) \in \mathsf{stats}(C)]$

$\mathsf{pn}(0) = \emptyset$
$\mathsf{pn}(I;\ C) = \mathsf{pn}(I) \cup \mathsf{pn}(C)$
$\mathsf{pn}(\{\, C \,\}) = \mathsf{pn}(C)$
$\mathsf{pn}(l, t : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x) = \{\mathsf{p}, \mathsf{q}\}$
$\mathsf{pn}(l, t : \mathsf{p} \rightsquigarrow \mathsf{q}.x) = \{\mathsf{q}\}$
$\mathsf{pn}(l, t : \mathsf{p} \to \mathsf{q}[L]) = \{\mathsf{p}, \mathsf{q}\}$
$\mathsf{pn}(l, t : \mathsf{p} \rightsquigarrow \mathsf{q}[L]) = \{\mathsf{q}\}$
$\mathsf{pn}(l, t : \mathsf{val}\ \mathsf{p}.x = e) = \{\mathsf{p}\}$
$\mathsf{pn}(l, t : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) =$
$\quad \{\mathsf{p}\} \cup \mathsf{pn}(C_1) \cup \mathsf{pn}(C_2)$
$\mathsf{pn}(l, t : X(\overline{\mathsf{p}}, \overline{a})) = \overline{\mathsf{p}}$
$\mathsf{pn}(l, t : \overline{\mathsf{q}}.\, X(\overline{\mathsf{p}}, \overline{a})\,\{\, C \,\}) = \overline{\mathsf{p}}$
$\mathsf{pn}(v@\mathsf{p}) = \{\mathsf{p}\}$
$\mathsf{pn}(\mathsf{p}.x) = \{\mathsf{p}\}$

Figure 9 Well-formedness (representative rules)

▶ **Theorem 2** (Deadlock-Freedom). *If* $\langle C,\ \Sigma,\ K \rangle$ *is well-formed, then either* $C \equiv 0$ *or* $\langle C,\ \Sigma,\ K \rangle \xrightarrow{\mathsf{p}} \langle C',\ \Sigma',\ K' \rangle$ *for some* $\mathsf{p}, C', \Sigma', K'$.

Proof. By induction on the structure of C, making use of the full definition of well-formedness in Appendix A. In each case, we observe the first instruction I of C can always be executed. For instance, if $I \equiv l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x$ then the C-Send rule can be applied because well-formedness implies e has no free variables. If $I \equiv l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$, there must be a message $(l, \tau, v) \in K(\mathsf{q})$ because the configuration is well-formed. The other cases follow similarly. ◀

We end this section with a formalization of communication integrity. Consider the buggy execution in Figure 2: in a model without integrity keys, the execution reaches a configuration

$$\langle\ \mathsf{s} \rightsquigarrow \mathsf{c}.txt;\ \mathsf{s} \rightsquigarrow \mathsf{c}.key;\ \ldots,\ \Sigma,\ \mathsf{c} \mapsto v_{key}, v_{txt}\ \rangle,$$

where v_{key} is the value produced by $\mathsf{ks}.getKey()$ and v_{txt} is the value produced by $\mathsf{cs}.getText()$. A CIV occurs if the configuration can make a transition that consumes $\mathsf{s} \rightsquigarrow \mathsf{c}.txt$ and v_{key} together, binding $\mathsf{c}.txt$ to v_{key}. We therefore want to ensure:

- There is only one way a communication-in-progress instruction can be consumed; and
- The instruction is consumed together with the correct message.

▶ **Definition 3** (Send/receive transitions). *A* send transition $\langle C,\ \Sigma,\ K \rangle \xrightarrow{\mathsf{p}} \langle C',\ \Sigma',\ K' \rangle$ *is a transition with a derivation that ends with an application of* C-Send. *Likewise, a* receive transition *is a transition with a derivation that ends with* C-Recv.

▶ **Theorem 4** (Communication Integrity). *Let* $e = c_0 \xrightarrow{\mathsf{p}_1} \cdots \xrightarrow{\mathsf{p}_{k+1}} c_{k+1}$ *be an execution ending with a send transition* $c_k \xrightarrow{\mathsf{p}} c_{k+1}$, *which produces instruction* $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ *and message* m. *Let* $e' = c_0 \xrightarrow{\mathsf{p}_1} \cdots \xrightarrow{\mathsf{p}_n} c_n$ $(n > k)$ *be an execution extending* e, *where* $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ *has not yet been consumed. Then there is at most one receive transition* $c_n \xrightarrow{\mathsf{q}} c_{n+1}$ *consuming* $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$. *Namely, it is the transition that consumes* $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ *and* m *together.*

Proof. By definition of C-Send, m has the form (l, τ, v). By definition of C-Recv, if there exists a transition $c_n \to c_{n+1}$ that consumes $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$, then the transition also consumes a message (l, τ, v'), for some v'. It therefore suffices to show the message (l, τ, v') is unique and that $v' = v$. This follows by induction on the length m of the extension:

- *Base case:* Well-formedness implies there is no message (l, τ, v') in c_k. Hence the message (l, τ, v) in c_{k+1} is unique.
- *Induction step:* Observe that the transition $c_m \to c_{m+1}$ cannot remove (l, τ, v); this would require consuming $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$, which cannot happen in e' by hypothesis. Also observe that the transition cannot add a new message with integrity key (l, τ); this would require consuming an instruction $l, \tau : \mathsf{p}'.e \to \mathsf{val}\ \mathsf{q}.x'$, which cannot exist in c_m by well-formedness. Hence (l, τ, v) is unique in c_{m+1}.

◀

4 Process Model and Endpoint Projection

4.1 Syntax

Figure 10 presents the syntax for out-of-order processes. A term $\mathsf{p}[P]$ is a process named p with behavior P. Networks, ranged over by N, M, are parallel compositions of processes. Compared to prior work, certain process instructions need to be annotated with integrity

$$
\begin{array}{rlll}
\mathscr{P} & ::= \{X_i(\overline{\mathsf{p}}_i, \overline{x}_i) = C_i\}_{i\in\mathcal{I}} & \textit{(decls)} & \\
P, Q & ::= I;\ P & \textit{(seq)} & \mid \{\, P \,\} \quad \textit{(block)} \\
& \mid 0 & \textit{(end)} & \\
I & ::= \mathsf{p}\,!_{l,t}\, e & \textit{(send)} & \mid ?_{l,t}\, x \quad \textit{(receive)} \\
& \mid \mathsf{val}\ x = e & \textit{(expr)} & \mid \mathsf{p} \oplus_{l,t} L \quad \textit{(choice)} \\
& \mid \&\{(l_i, \tau, L_i) \Rightarrow P_i\}_{i\in\mathcal{I}} & \textit{(branch)} & \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \quad \textit{(cond)} \\
& \mid l, t : X(\overline{\mathsf{p}}, \overline{a}) & \textit{(call)} & \\
e & ::= f(\overline{e}) & \textit{(app)} & \mid a \quad \textit{(atom)} \\
a & ::= x & \textit{(var)} & \mid v \quad \textit{(val)} \\
N, M & ::= \mathsf{p}[P] & \textit{(proc)} & \mid (N \mid M) \quad \textit{(par)}
\end{array}
$$

Figure 10 Syntax for out-of-order processes

keys (for instance, message send $\mathsf{p}\,!_{l,t}\, e$ and procedure call $l, t : X(\overline{\mathsf{p}}, \overline{a})$). In addition, when receiving a message it is no longer necessary to specify a sender—it suffices to write $?_{l,t}\, x;\ P$ instead of the more traditional $\mathsf{p}\,?_{l,t}\, x;\ P$—because integrity keys functionally determine the variable to which the message payload should be bound.

4.2 Semantics

The semantics for out-of-order processes appears in Figure 11. It is a labelled transition system on *process configurations* $\langle\ N,\ \Sigma,\ K\ \rangle$, where N is a network and Σ, K have the same meaning as in Section 3.2. We also assume an implicit set of procedure declarations $\mathscr{P}$.

The transition rules of Figure 11 are similar to prior work. P-Send adds a message (l, τ, v) to the undelivered messages of q, whereas P-Recv removes the message and substitutes it into the body of the process. Similarly, P-Select adds (l, τ, L) to the message set and P-OnSelect selects a branch from the set of options $\&\{(l_j, \tau_j, L_j) \Rightarrow P_j\}_{j\in\mathcal{J}}$. P-Call invokes a procedure, locally computing the next token and substituting the body of the procedure into the process. Rules P-Compute, P-If, and P-Par are standard.

The key novelty of out-of-order processes is the P-Delay rule, which allows a process to perform instructions in *any* order, up to data- and control-dependencies. The latter implies processes cannot evaluate instructions nested within an if or $\&$-expression.

4.3 Endpoint Projection

Figure 13 defines the *endpoint projection (EPP)* $[\![C]\!]$ of a choreography C, translating it into a network. The rules follow from simple modifications to the textbook definition of EPP [26]. Projecting a conditional on a process that does not evaluate the guard uses the auxiliary partial operator $\sqcup$, which produces a term that can react to the different branches by receiving different selections (this is standard).

Figure 12 shows networks projected from the choreographies of Figure 7. Notice the choreographic procedures *BuyItem* and *StreamIt* are each split into two process procedures—one for each role. Communications in the choreography are, as usual, projected into send and receive instructions. Conditionals in the choreography are projected into an if-instruction at one process and a branch-instruction at the other processes awaiting its decision.

Below we formulate the hallmark *EPP Theorem*, which states that a choreography C and its projection $[\![C]\!]$ evolve in lock-step, up to the usual ($\sqsupseteq$) relation from Montesi [26] (given

$$\frac{\Sigma(\mathsf{p}) \vdash e \Downarrow (v, \sigma) \qquad M = K(\mathsf{q}) \uplus \{(l, \tau, v)\}}{\langle\, \mathsf{p}[\mathsf{q}\,!_{l,\tau}\, e;\ P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[P],\ \Sigma[\mathsf{p} \mapsto \sigma],\ K[\mathsf{q} \mapsto M] \,\rangle}\ \textsc{P-Send}$$

$$\frac{(l, \tau, v) \in K(\mathsf{q}) \qquad M = K(\mathsf{q}) \setminus \{(l, \tau, v)\}}{\langle\, \mathsf{q}[?_{l,\tau}\, x;\ Q],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{q}} \langle\, \mathsf{q}[Q[x \mapsto v]],\ \Sigma,\ K[\mathsf{q} \mapsto M] \,\rangle}\ \textsc{P-Recv}$$

$$\frac{M = K(\mathsf{q}) \cup \{(l, \tau, L)\}}{\langle\, \mathsf{p}[\mathsf{q} \oplus_{l,\tau} L;\ P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[P],\ \Sigma,\ K[\mathsf{q} \mapsto M] \,\rangle}\ \textsc{P-Select}$$

$$\frac{K(\mathsf{q}) = \{(l_i, \tau, L_i)\} \cup M \qquad i \in \mathcal{I}}{\langle\, \mathsf{q}[\&\{(l_j, \tau, L_j) \Rightarrow Q_j\}_{j \in \mathcal{I}};\ Q],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{q}} \langle\, \mathsf{q}[\{Q_i\};\ Q],\ \Sigma,\ K[\mathsf{q} \mapsto M] \,\rangle}\ \textsc{P-OnSelect}$$

$$\frac{\Sigma(\mathsf{p}) \vdash e \Downarrow (v, \sigma)}{\langle\, \mathsf{p}[\mathsf{val}\ x = e;\ P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[P[x \mapsto v]],\ \Sigma[\mathsf{p} \mapsto \sigma],\ K \,\rangle}\ \textsc{P-Compute}$$

$$\frac{\Sigma(\mathsf{p}) \vdash e \Downarrow v \qquad \text{if } v = \mathsf{true} \text{ then } i = 1 \text{ else } i = 2}{\langle\, \mathsf{p}[\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2;\ P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[\{P_i\};\ P],\ \Sigma,\ K \,\rangle}\ \textsc{P-If}$$

$$\frac{\langle\, \mathsf{p}[P_1],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[P_1'],\ \Sigma',\ K' \,\rangle}{\langle\, \mathsf{p}[\{P_1\};\ P_2],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[\{P_1'\};\ P_2],\ \Sigma',\ K' \,\rangle}\ \textsc{P-Block}$$

$$\frac{\langle\, \mathsf{p}[P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[P'],\ \Sigma',\ K' \,\rangle}{\langle\, \mathsf{p}[I;\ P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[I;\ P'],\ \Sigma',\ K' \,\rangle}\ \textsc{P-Delay}$$

$$\frac{(X(\overline{\mathsf{q}}, \overline{y}) = Q) \in \mathscr{P} \qquad \mathsf{nextToken}(l, \tau) = \tau'}{\langle\, \mathsf{p}[l, \tau : X(\overline{\mathsf{p}}, \overline{a});\ P],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, \mathsf{p}[\{Q[\overline{\mathsf{q}}, \overline{y}, \mathsf{t} \mapsto \overline{\mathsf{p}}, \overline{a}, \tau']\};\ P],\ \Sigma,\ K \,\rangle}\ \textsc{P-Call}$$

$$\frac{\langle\, N,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, N',\ \Sigma',\ K' \,\rangle}{\langle\, N \mid M,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, N' \mid M,\ \Sigma',\ K' \,\rangle}\ \textsc{P-Par}$$

Figure 11 Semantics for out-of-order processes

(a)

$$
\begin{aligned}
&\mathit{BuyItem}_1(\mathsf{b}) = \\
&\quad ?_{1,\mathsf{t}}\ \mathit{itemID}; \\
&\quad \mathsf{val}\ \mathit{item?} = \mathit{sell}(\mathit{itemID}); \\
&\quad \mathsf{b}\,!_{3,\mathsf{t}}\ \mathit{item?} \\
&\mathit{BuyItem}_2(\mathsf{s},\ \mathit{itemID}) = \\
&\quad \mathsf{s}\,!_{1,\mathsf{t}}\ \mathit{itemID}; \\
&\quad ?_{3,\mathsf{t}}\ \mathit{item?} \\
&\mathsf{seller}[4, \tau_0 : \mathit{BuyItem}_1(\mathsf{buyer}_1); \\
&\quad 5, \tau_0 : \mathit{BuyItem}_1(\mathsf{buyer}_2)] \mid \\
&\mathsf{buyer}_1[4, \tau_0 : \mathit{BuyItem}_2(\mathsf{seller}, 123)] \mid \\
&\mathsf{buyer}_2[5, \tau_0 : \mathit{BuyItem}_2(\mathsf{seller}, 543)]
\end{aligned}
$$

(b)

$$
\begin{aligned}
&\mathit{StreamIt}_1(\mathsf{c}) = \\
&\quad \mathsf{c}\,!_{1,\mathsf{t}}\ \mathit{produce}(); \\
&\quad \mathsf{if}\ (\mathit{itemsLeft}() > 0)\ \mathsf{then} \\
&\qquad \mathsf{c} \oplus_{4,\mathsf{t}} \textsc{More};\ 5, \mathsf{t} : \mathit{StreamIt}_1(\mathsf{c}) \\
&\quad \mathsf{else}\ \mathsf{c} \oplus_{6,\mathsf{t}} \textsc{Done} \\
&\mathit{StreamIt}_2(\mathsf{p}) = \\
&\quad ?_{1,\mathsf{t}}\ x;\ \mathsf{val}\ z = \mathit{consume}(x); \\
&\quad \&\{(4, \mathsf{t}, \textsc{More}) \Rightarrow 5, \mathsf{t} : \mathit{StreamIt}_2(\mathsf{p}), \\
&\qquad (6, \mathsf{t}, \textsc{Done}) \Rightarrow 0\} \\
&\mathsf{p}_1[7, \tau_0 : \mathit{StreamIt}_1(\mathsf{c})] \mid \\
&\mathsf{p}_2[8, \tau_0 : \mathit{StreamIt}_1(\mathsf{c})] \mid \\
&\mathsf{c}[7, \tau_0 : \mathit{StreamIt}_2(\mathsf{p});\ 8, \tau_0 : \mathit{StreamIt}_2(\mathsf{p})]
\end{aligned}
$$

Figure 12 Processes projected from Figure 7.

in Figure 14). Importantly, we update the theorem to restrict our attention to *well-formed* networks and choreographies. We say that a network N is well-formed if the keys in each process are distinct, i.e., $\mathsf{keys}(P)$ is distinct for each $\mathsf{p}[P]$ in N ($\mathsf{keys}(P)$ is given in Figure 14). The restriction allows us to ignore processes such as $\mathsf{p}[\&\{(1, \tau, L) \Rightarrow P_1\};\ \&\{(1, \tau, L) \Rightarrow P_2\}]$, which could only be projected from a choreography where two distinct instructions have the same integrity key $(1, \tau)$. This leads to the following lemma:

▶ **Lemma 5.** *Let C, Q be well-formed. If $Q \sqsupseteq [\![C]\!]_{\mathsf{q}}$ then $\mathsf{keys}(Q) \supseteq \mathsf{keys}_{\mathsf{q}}(C)$, where*

$$\mathsf{keys}_{\mathsf{q}}(C) = [(l, t) \mid (l, t : \mathsf{p} \rightsquigarrow \mathsf{q}.x) \in \mathsf{stats}(C)], [(l, t) \mid (l, t : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x) \in \mathsf{stats}(C)].$$

The key difficulty of proving the EPP Theorem was finding the right definition of well-formedness (Theorems 1 and 4). With the definition established, the entire proof follows directly from textbook induction principles (*c.f.* [26]). We sketch the proof in Appendix B.

▶ **Theorem 6** (EPP Theorem). *Let $\langle\, C,\ \Sigma,\ K \,\rangle$ be a well-formed configuration.*

1. *(Completeness) If $\langle\, C,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, C',\ \Sigma',\ K' \,\rangle$ then $\langle\, [\![C]\!],\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, N',\ \Sigma',\ K' \,\rangle$ for some well-formed N' where $N' \sqsupseteq [\![C']\!]$.*
2. *(Soundness) If $\langle\, N,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{r}} \langle\, N',\ \Sigma',\ K' \,\rangle$ for some well-formed N where $N \sqsupseteq [\![C]\!]$, then $\langle\, C,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{p}} \langle\, C',\ \Sigma',\ K' \,\rangle$ for some C' where $N' \sqsupseteq [\![C']\!]$.*

5 A Non-Blocking Communication API for Choral

In this section, we show how the ideas in O_3 can be applied in practice. To this end, we consider Choral [16]: a state-of-the-art choreographic programming language based on Java. Choral is designed to support real-world programming and interoperate with Java, so it is much more sophisticated than our minimalistic theory. Data locations in Choral are lifted to the type level and communication is expressed by invoking methods of *channel* objects.

$$[\![\mathscr{C}]\!] = \bigcup_{i\in\mathcal{I}} [\![X_i(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C_i]\!]$$

$$[\![X_i(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C_i]\!] = \{X_{i,j}(\overline{\mathsf{p}} \setminus \mathsf{p}_j, [\![\overline{\mathsf{p}.x}]\!]_{\mathsf{p}_j}) = [\![C_i]\!]_{\mathsf{p}_j} \mid \overline{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n,\ j \le n\}$$

$$[\![l, t : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x;\ C]\!]_\mathsf{r} = \begin{cases} \mathsf{q}\,!_{l,t}\, e;\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ ?_{l,t}\, x;\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : \mathsf{p} \rightsquigarrow \mathsf{q}.x;\ C]\!]_\mathsf{r} = \begin{cases} ?_{l,t}\, x;\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : \mathsf{val}\ \mathsf{p}.x = e;\ C]\!]_\mathsf{r} = \begin{cases} \mathsf{val}\ x = [\![e]\!]_\mathsf{r};\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : \mathsf{p} \to \mathsf{q}[L];\ C]\!]_\mathsf{r} = \begin{cases} \mathsf{q} \oplus_{l,t} [\![e]\!]_\mathsf{r};\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \&\{(l, t, L) \Rightarrow [\![C]\!]_\mathsf{r}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : \mathsf{p} \rightsquigarrow \mathsf{q}[L];\ C]\!]_\mathsf{r} = \begin{cases} \&\{(l, t, L) \Rightarrow [\![C]\!]_\mathsf{r}\} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;\ C]\!]_\mathsf{r} = \begin{cases} \mathsf{if}\ [\![e]\!]_\mathsf{r}\ \mathsf{then}\ [\![C_1]\!]_\mathsf{r}\ \mathsf{else}\ [\![C_2]\!]_\mathsf{r};\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C_1]\!]_\mathsf{r} \sqcup [\![C_2]\!]_\mathsf{r};\ [\![C]\!]_\mathsf{r} & \text{if } \mathsf{r} \in \mathsf{pn}(C_1, C_2) \setminus \mathsf{p} \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : X_i(\overline{\mathsf{p}}, \overline{a});\ C]\!]_\mathsf{r} = \begin{cases} l, t : X_{i,j}(\overline{\mathsf{p}} \setminus \mathsf{p}_j, [\![\overline{a}]\!]_{\mathsf{p}_j});\ [\![C]\!]_{\mathsf{p}_j} & \text{if } \mathsf{r} = \mathsf{p}_j \text{ where } \overline{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_n \\ [\![C]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![l, t : \overline{\mathsf{q}}.\, X_i(\overline{\mathsf{p}}, \overline{a})\,\{\,C_1\,\};\ C_2]\!]_\mathsf{r} = \begin{cases} l, t : X_{i,j}(\overline{\mathsf{p}} \setminus \mathsf{p}_j, [\![\overline{a}]\!]_{\mathsf{p}_j});\ [\![C_2]\!]_{\mathsf{p}_j} & \text{if } \mathsf{r} \in \overline{\mathsf{q}} \text{ and } \mathsf{r} = \mathsf{p}_j \\ [\![C_1;\ C_2]\!]_r & \text{if } \mathsf{r} \in \overline{\mathsf{p}} \setminus \overline{\mathsf{q}} \\ [\![C_2]\!]_\mathsf{r} & \text{otherwise} \end{cases}$$

$$[\![\{\,C_1\,\};\ C_2]\!]_\mathsf{r} = \{[\![C_1]\!]_\mathsf{r}\};[\![C_2]\!]_\mathsf{r} \qquad [\![v@\mathsf{p}]\!]_\mathsf{r} = \begin{cases} v & \text{if } \mathsf{r} = \mathsf{p} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![a_1, \ldots, a_n]\!]_\mathsf{r} = [\![a_1]\!]_\mathsf{r}, \ldots, [\![a_n]\!]_\mathsf{r}$$

$$[\![f(e_1, \ldots, e_n)]\!]_\mathsf{r} = f([\![e_1]\!]_\mathsf{r}, \ldots, [\![e_n]\!]_\mathsf{r}) \qquad [\![\mathsf{p}.x]\!]_\mathsf{r} = \begin{cases} x & \text{if } \mathsf{r} = \mathsf{p} \\ \bot & \text{otherwise} \end{cases}$$

$$(\&\{(l_i, \tau_i, L_i) \Rightarrow P_i\}_{i\in\mathcal{I}}) \sqcup (\&\{(l_j, \tau_j, L_j) \Rightarrow P_j\}_{j\in\mathcal{J}}) = \&\{(l_k, \tau_k, L_k) \Rightarrow P_k\}_{k\in\mathcal{I}\cup\mathcal{J}} \quad \text{if } \{L_i : i \in \mathcal{I}\} \# \{L_j : j \in \mathcal{J}\}$$

Figure 13 Endpoint projection

$$
\begin{array}{ll}
0 \sqsupseteq 0 & \\
(P_1;\, P_2) \sqsupseteq (Q_1;\, Q_2) & \text{if } P_i \sqsupseteq Q_i \text{ for } i = 1, 2 \\
(\mathsf{if}\, e\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2) \sqsupseteq (\mathsf{if}\, e\, \mathsf{then}\, Q_1\, \mathsf{else}\, Q_2) & \text{if } P_i \sqsupseteq Q_i \text{ for } i = 1, 2 \\
I_1 \sqsupseteq I_2 & \text{if } I_1 = I_2 \text{ or } I_1 = I_1 \sqcup I_2
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{keys}(0) = \epsilon & \\
\mathsf{keys}(I;\, P) = \mathsf{keys}(I), \mathsf{keys}(P) & \\
\mathsf{keys}(\mathsf{p}\,!_{l,t}\, e) = (l, t) & \\
\mathsf{keys}(?_{l,t}\, x) = (l, t) & \\
\mathsf{keys}(\mathsf{val}\ x = e) = \epsilon & \\
\mathsf{keys}(\mathsf{p} \oplus_{l,t} L) = (l, t) & \\
\mathsf{keys}(\&\{(l_i, \tau_i, L_i) \Rightarrow P_i\}_{i \in \mathcal{I}}) = & \\
\quad [(l_i, \tau_i) \mid i \in \mathcal{I}], [\mathsf{keys}(P_i) \mid i \in \mathcal{I}] & \\
\mathsf{keys}(\mathsf{if}\, e\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2) = \mathsf{keys}(P_1), \mathsf{keys}(P_2) & \\
\mathsf{keys}(l, t : X(\overline{\mathsf{p}}, \overline{a})) = (l, t) & \\
\mathsf{keys}(\{P_1\};\, P_2) = \mathsf{keys}(P_1), \mathsf{keys}(P_2) &
\end{array}
$$

Figure 14 Auxiliary definitions for the EPP Theorem ($\sqsupseteq$ and keys)

```
1  public class ConcurrentSend@(KS, CS, S, C) {
2      public void start(
3          String@KS key, String@CS txt, Client@C client,
4          Token@(KS, CS, S, C) tok,
5          AsyncChannel@(KS, S) ch1, AsyncChannel@(CS, S) ch2, AsyncChannel@(S, C) ch3
6      ) {
7          // Services send data to the server.
8          CompletableFuture@S keyS = ch1.fcom(key, 1@(KS,S), tok);
9          CompletableFuture@S txtS = ch2.fcom(txt, 2@(CS,S), tok);
10
11         // Server forwards data to the client.
12         ch3.fcom(keyS, 3@(S,C), tok)
13             .thenAccept(client::decrypt);
14         ch3.fcom(txtS, 4@(S,C), tok)
15             .thenAccept(client::display);
16     }
17 }
```

Figure 15 An implementation of the choreography in Figure 2 using Choral and the Ozone API.

Choral's intended programming model consists of sequential processes that block to receive messages. However, to improve performance programmers can use Java's `CompletableFuture` API, thereby introducing intraprocess concurrency and out-of-order execution. This breaks the programming model and introduces CIVs (cf. Section 2) that could cause crashes or silent memory corruption. Motivated by our formal model, we developed *Ozone*: an API for Choral programmers to safely mix choreographies with futures. In the remainder of this section, we introduce Choral and Ozone and we show how programmers can mix choreographies with futures achieve significant speedups in practical applications.

5.1 Concurrent Messages

We introduce the Ozone API with an implementation of the choreographic procedure from Figure 2. The implementation is shown in Figure 15, which defines a class called `ConcurrentSend` parameterized by four roles (i.e. process parameters): `KS`, `CS`, `S`, and `C`. In this class, the `start` method implements the procedure itself. As in our formal model, the procedure is parameterized by distributed data: On line 3, parameter `key` is a `String`

located at KS; txt is a String located at CS; and client is a Client object at C, representing the client's user interface. The start procedure is also parameterized by session tokens, which we introduced in Figure 15, on line 4. The parameter Token@(KS, CS, S, C) tok is syntactic sugar for the parameter *list* Token@KS tok_KS, ..., Token@C tok_C.[3] The last three parameters on line 5 are *channels*. In Choral, channels are used to communicate data from one role to another. If ch is a channel of type Channel@(A,B)<T> and e is an expression of type T@A, then the expression ch.com(e) is a communication that produces a value of type T@B.

Our main contribution in the Ozone API is a custom channel AsyncChannel@(A,B)<T> with a method fcom for safely communicating data with non-blocking semantics. The fcom method is similar to com, but with the following differences:

- Whereas com takes one argument, fcom takes three: a payload, a line number, and a session token. The latter two arguments form an integrity key, of which both the sender and receiver have a copy.
- When the receiver B executes a com instruction, its thread becomes blocked until the value (of type T@B) has been delivered. In contrast, fcom creates a Java *future* (of type CompletableFuture@B<T>) which is a placeholder at B that will hold a value of type T once the message is delivered. Instead of blocking, fcom immediately returns that future to the calling thread. The thread can then assign a callback to handle the message and proceed with other useful work.

Lines 8 and 9 of Figure 15 show fcom being used to transport key and txt to the server S. The expression 1@(KS, S) is sugar for the list 1@KS, 1@S and we assume the replicated value tok is expanded into the list tok_KS, tok_S. Thus both sender and receiver pass integrity keys as arguments to fcom.

Lines 12-15 of Figure 15 show how the server S and client C use the future values. On line 12, the server uses an overloaded version of fcom that takes CompletableFuture@S instead of T@S. The method assigns to the future a callback, which forwards the result to the client once the future has been completed. The result of fcom on line 12 is a CompletableFuture@C, to which the client binds a callback on line 13: when the key from S finally arrives at C, the client will proceed to invoke the method client.decrypt with the key as an argument. Lines 14 and 15 do the same, but with the value of txt. As we will see below, the values of key and txt can arrive at the client in any order, so the callbacks on lines 13 and 15 can execute in any order—even in parallel.

5.1.1 Endpoint projection

By running the Choral compiler, ConcurrentSend@(KS,CS,S,C) is *projected* to generate four Java classes, shown in Figure 16. Each class implements the behavior of its corresponding role. For example, ConcurrentSend_KS implements the behavior of KS. Its start method is parameterized by: key, which corresponds to the key in Figure 15; tok_KS, the copy of the token tok belonging to KS; and ch1, a channel endpoint that connects KS to S. Following the reasoning in Figure 13, these behaviors will not exhibit deadlocks or communication integrity errors when composed (assuming the implementations of Choral and Ozone are correct).

Let us see how integrity keys prevent CIVs in Figure 16. Notice that the Choral instruction CompletableFuture@S keyS = ch1.fcom(key, 1@(KS,S), tok); on line 8 of Figure 15 is

[3] This syntactic sugar is provided for readability and is not currently supported by the Choral compiler. We will also use syntactic sugar for lambda expressions and omit obvious type annotations later in this section. Our actual implementation uses desugared versions of the syntax.

```
1  public class ConcurrentSend_KS {
2    public void start(
3      String key, Token tok_KS,
4      AsyncChannel ch1
5    ) {
6      ch1.fcom(key, 1, tok_KS);
7    }
8  }
9  public class ConcurrentSend_S {
10   public void start(
11     Token tok_S, AsyncChannel ch1,
12     AsyncChannel ch2, AsyncChannel ch3
13   ) {
14     CompletableFuture keyS =
15       ch1.fcom(1, tok_S);
16     CompletableFuture txtS =
17       ch2.fcom(2, tok_S);
18
19     ch3.com(keyS, 3, tok_S);
20     ch3.com(txtS, 4, tok_S);
21   }
22 }
```

```
23 public class ConcurrentSend_CS {
24   public void start(
25     String txt, Token tok_CS,
26     AsyncChannel ch2
27   ) {
28     ch2.fcom(txt, 2, tok_CS);
29   }
30 }
31
32 public class ConcurrentSend_C {
33   public void start(
34     Client client, Token tok_C,
35     AsyncChannel ch3
36   ) {
37     ch3.fcom(3, tok_C)
38        .thenAccept(client::decrypt);
39     ch3.fcom(4, tok_C)
40        .thenAccept(client::display);
41   }
42 }
```

Figure 16 Endpoint projection of Figure 15.

projected into two instructions:

1. `ch1.fcom(key, 1, tok_KS)` at the sender `KS`; and
2. `CompletableFuture keyS = ch1.fcom(1, tok_S)` at the receiver `S`.

The former instruction is parameterized by a payload and an integrity key and produces nothing. The latter instruction is parameterized only by an integrity key (with no payload) and produces a future. When `KS` sends `key` to `S`, it combines the payload with integrity key `(1, tok_KS)`. Dually, `S` creates a future that will only be completed when a message with the integrity key `(1, tok_S)` is received. Since `tok_KS` and `tok_S` have the same value, the send- and receive-operations are guaranteed to match.

On lines 14-17 of Figure 16, the server `S` sets listeners for `key` and `txt`. On lines 19-20, `S` schedules the values to be forwarded to `C`; notice that even with FIFO channels, `key` and `txt` may arrive in any order. Consequently, `S` may forward their values to `C` in any order. On lines 37-40 of Figure 16, the client creates futures to hold the values of `key` and `txt` and sets callbacks to be invoked when the values arrive. Here we see the importance of integrity keys: the client uses `(3, tok_C)` and `(4, tok_C)` to disambiguate the `key` message from the `txt` message. Without integrity keys, mixing Choral choreographies with Java Futures would be unsafe. As shown in Section 4.3, our solution is correct even when the underlying transport protocol can deliver messages out of order.

5.2 Procedure calls

Section 5.1 showed how the line numbers in an integrity key could prevent CIVs. We now briefly show how the *tokens* in an integrity key prevent *interprocedural* CIVs. Figure 17 depicts a choreography that invokes two instances of `ConcurrentSend2`: the first instance with client `C1`, and the second instance with client `C2`. On lines 12 and 16, the roles all compute fresh tokens for each procedure they're involved in, like in our formal model; the syntax `tok.nextToken( 0@(KS,CS,S,C1) )` is sugar for `tok_KS.nextToken(0@KS), ..., tok_C1.nextToken(0@C1)`, and the method `t.nextToken(l)` implements the function $\mathsf{nextToken}(l, t)$. These fresh tokens ensure that, even if messages from `KS` to `S` are

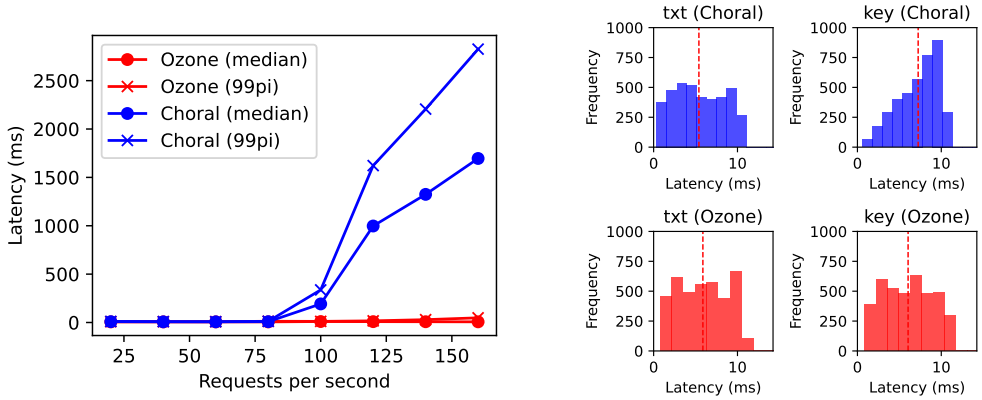
```
public class ConcurrentClients@(KS, CS, S, C1, C2) {
    public void start(
        AsyncChannel@(KS, S) ch1, AsyncChannel@(CS, S) ch2,
        AsyncChannel@(S, C1) ch3, AsyncChannel@(S, C2) ch4,
        KeyService@KS keyService, ContentService@CS contentService,
        Client@C1 client1, String@(KS, CS) clientID1,
        Client@C2 client2, String@(KS, CS) clientID2,
        Token@(KS, CS, S, C1, C2) tok
    ) {
        (new ConcurrentSend2()).start(ch1, ch2, ch3,
            keyService.getKey(clientID1), contentService.getContent(clientID1),
            client1, tok.nextToken( 0@(KS,CS,S,C1) ));

        (new ConcurrentSend2()).start(ch1, ch2, ch4,
            keyService.getKey(clientID2), contentService.getContent(clientID2),
            client2, tok.nextToken( 1@(KS,CS,S,C2) ));
    }
}
```

Figure 17 A Choral choreography invoking `ConcurrentSend2`.

```
public class ConcurrentClients_KS {
  public void start( ... ) {
    (new ConcurrentSend2_KS()).start(ch1,
        keyService.getKey(clientID1),
        tok.next(0));

    (new ConcurrentSend2_KS()).start(ch1,
        keyService.getKey(clientID2),
        tok.next(1));
  }
}
```

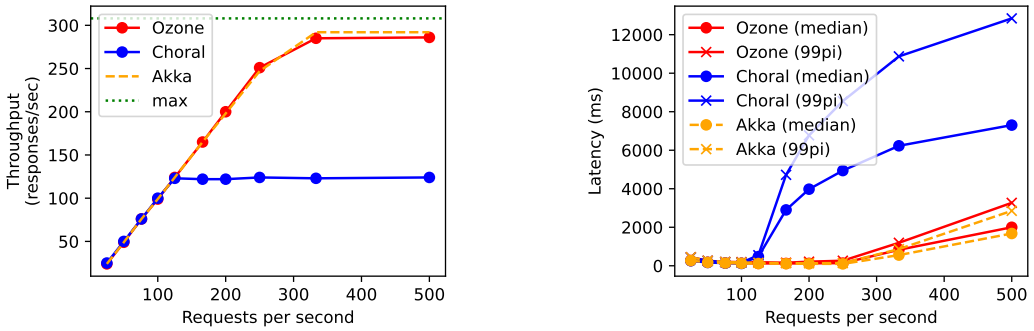
```
public class ConcurrentClients_S {
  public void start( ... ) {
    (new ConcurrentSend2_S()).start(
        ch1, ch2, ch3, tok.next(0));

    (new ConcurrentSend2_S()).start(
        ch1, ch2, ch3, tok.next(1));
  }
}
```

Figure 18 Endpoint projection of Figure 17 (representative examples).



(a) Concurrent producers latency (lower is better) **(b)** Concurrent senders latency (further left is better)

Figure 19 Microbenchmark



(a) Throughput (higher is better) **(b)** Latency (lower is better)

Figure 20 Model serving

delivered out of order there is no chance that messages from the first procedure invocation will be confused for messages from the second invocation.

5.3 Evaluation

We evaluated Ozone with microbenchmarks based on Figures 1 and 2 and with a model serving benchmark from Wang et al [33]. The experiments were carried out on a six-node Linux cluster, with two Intel Xeon Gold 6130 CPUs and 384 GB of memory per node and an average bandwidth of 15 Gbps.

The first microbenchmark is a version of Figure 1 from the introduction. Each producer iteratively invokes the choreography at a fixed rate and, in response to each request, the server simulates computation by sleeping for 0–5 milliseconds. Figure 19a shows the median and 99th percentile latency for server responses to worker requests. In the Choral implementation, the server quickly becomes a bottleneck because of its fixed processing order: a request from p_1 must be handled before a request from p_2, and both requests must be handled for the i-th iteration before they can be handled for the $(i+1)$-th iteration. In the Ozone implementation, requests from different producers can be handled out of order and producers can start a new instance of the choreography without waiting for the second one to complete. Consequently

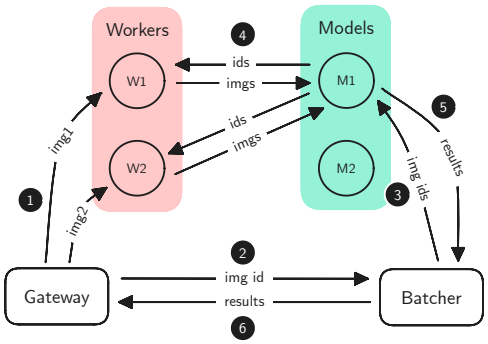


Figure 21 Architecture for the image classification pipeline.

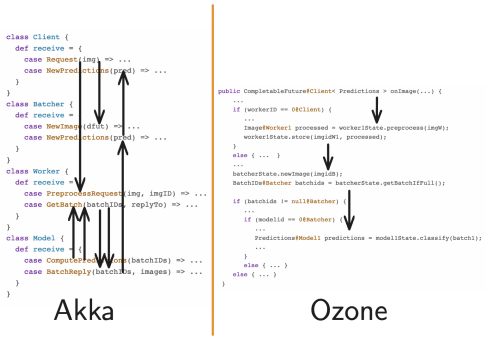


Figure 22 Control flow comparison between hand-written Akka processes and Ozone.

the server spends less time waiting for requests, so it can handle much higher request rates.

The second microbenchmark is a version of Figure 2 from Section 2, in which the server sends messages to the microservices *ks* and *cs* and forwards their responses to the client. Each microservice takes 0–5 milliseconds to compute its response. The latency histogram for the Choral implementation (top) shows how the time for the client to receive `txt` depends on the time to compute `txt`, but the time to receive `key` depends on *both* the time to compute `txt` and the time to compute `key`. In contrast, the Ozone implementation (bottom) allows the server to forward `key` to the client without waiting for `txt`—thereby reducing the average latency for `key` by more than 30%.

To measure the impact of Ozone on a realistic application, we ported the image classification pipeline of Wang et al [33] to Choral (Figure 21). In this pipeline, images are received by a Gateway that performs load balancing and forwards the images to a pair of Worker services for preprocessing. A Batcher service collects requests and sends them as a batch to a Model service, which fetches the processed images and performs image classification. This architecture allows applications to harness *intra*-GPU parallelism by increasing the batch size (at the cost of latency) and *inter*-GPU parallelism by increasing the number of Model services. Figure 20 shows the performance for Choral and Ozone implementations of the pipeline, using sleeps to simulate computation. The plots also show the performance of an implementation in the *Akka* actor framework [1] and the theoretical maximum throughput of the Model services.

The Choral implementation has a bottleneck: after the Batcher sends work to a Model, it waits for the Model's response and becomes blocked. In the Ozone implementation, the Batcher binds the Model service's response to a `CompletableFuture` and continues receiving requests from the Gateway. Consequently, the throughput and latency for the Ozone implementation can scale with the number of requests until both Models become saturated with work. Figure 20 shows our library scales similarly to hand-written reactive processes in Akka, though the latter perform slightly better under high load because the Akka framework is heavily optimized to handle network congestion; these same optimizations can be applied to Choral, but they are orthogonal to our present work. We conclude that our methodology can achieve good performance while providing the benefits of choreographic programming: (i) absence of bugs like deadlocks and mismatched communications (e.g., sending a message with the wrong type or at the wrong time) [16, 26, 23]; (ii) and improved readability, since control flow is easier to follow in choreographies than in processes (see Figure 22), as discussed in [31, 21, 23, 16].

6 Related Work

In early choreographic languages, the sequencing operator $I;\ C$ had strict sequential semantics; concurrency could only be introduced via an explicit parallel operator $C \,||\, C'$ [29, 22, 7]. Explicit parallelism was later replaced by a relaxed sequencing operator $I;\ C$ that would allow instructions in C to be evaluated before I under certain conditions [8]. This presents the benefits of offering a simple syntax for choreographies and, at the same time, automatically inferring what can be safely executed concurrently. For these reasons, relaxed sequencing has been adopted in most recent works on choreographic programming (e.g., [18, 16, 17, 13, 20, 21]) and its textbook presentation [26]. Our present work makes the sequencing operator even more relaxed, allowing all instructions to be executed out of order, up to data- and control-dependency. To the best of our knowledge, our model is the first to support non-FIFO communication in the setting of choreographic programming.

Our work is closely related to choreographic *multicoms*: sets of communications that can be executed out of order, up to data dependency [12]. However, multicoms do not allow *computation* to be performed out of order, as in Figure 1c. Multicoms therefore do not need to address the communication integrity problem, which we focus on in this work. Relatedly, previous work investigated modeling asynchronous communication by making send actions non-blocking [8, 19, 10, 27, 18, 26], but none of considered non-blocking receive. Thus, they are not expressive enough to capture the behaviors that we are interested in here.

In terms of expressivity, there is some overlap between our model and *nondeterministic choreographies* [26], which use an explicit *choreographic choice* operator $C +_{\mathsf{p}} C'$. Nondeterministic choreographies can implement the execution in Figure 1c with:

$$\begin{pmatrix} \mathsf{buyer}_1.id \to \mathsf{val}\ \mathsf{seller}.id_1; \\ \mathsf{buyer}_2.id \to \mathsf{val}\ \mathsf{seller}.id_2; \\ \cdots \end{pmatrix} +_{\mathsf{seller}} \begin{pmatrix} \mathsf{buyer}_2.id \to \mathsf{val}\ \mathsf{seller}.id_2; \\ \mathsf{buyer}_1.id \to \mathsf{val}\ \mathsf{seller}.id_1; \\ \cdots \end{pmatrix}$$

Figure 2 can also be expressed with nondeterministic choreographies:

$$\begin{pmatrix} 1 : \mathsf{cs}.getText() \to \mathsf{val}\ \mathsf{s}.txt; \\ 2 : \mathsf{s} \to \mathsf{c}[\textsc{TxtFirst}]; \\ 3 : \mathsf{s}.txt \to \mathsf{val}\ \mathsf{c}.txt; \\ 4 : \mathsf{c}.display(\mathsf{c}.txt); \\ 5 : \mathsf{ks}.getKey() \to \mathsf{val}\ \mathsf{s}.key; \\ 6 : \mathsf{s}.key \to \mathsf{val}\ \mathsf{c}.key; \\ 7 : \mathsf{c}.decrypt(\mathsf{c}.key) \end{pmatrix} +_{\mathsf{s}} \begin{pmatrix} 8 : \mathsf{ks}.getKey() \to \mathsf{val}\ \mathsf{s}.key; \\ 9 : \mathsf{s} \to \mathsf{c}[\textsc{KeyFirst}]; \\ 10 : \mathsf{s}.key \to \mathsf{val}\ \mathsf{c}.key; \\ 11 : \mathsf{c}.decrypt(\mathsf{c}.key); \\ 12 : \mathsf{cs}.getText() \to \mathsf{val}\ \mathsf{s}.txt; \\ 13 : \mathsf{s}.txt \to \mathsf{val}\ \mathsf{c}.txt; \\ 13 : \mathsf{c}.display(\mathsf{c}.txt) \end{pmatrix}$$

Compared to O_3, these implementations are much larger because they require programmers to statically encode all desired schedules. One can easily forget to include some schedules or encode them incorrectly: for instance, if one moved line 12 up to line 10 above, it would eliminate the extra concurrency that was gained by receiving the messages out of order. Thus, our approach is more robust and simpler for the programmer.

On the other hand, nondeterministic choreographies can express some programs that our model cannot. For example, choreographic choice can assign different variable names to messages, according to their arrival order. Other choreographic languages include nondeterministic operators [22, 6], but they do not support computation (a requirement for choreographic programming) or recursion.

Choral is arguably the most powerful implementation of choreographic programming to date, but there are also others that target, e.g., Haskell, Java, Jolie, and Rust [8, 9, 28, 31, 21].

We believe that implementing the out-of-order semantics of O_3 in these languages is possible, too. However, it would likely require more work that touches also the implementation of EPP because, differently from Choral, these languages do not support user-defined communication primitives. Since Choral is more expressive than all other current choreographic programming languages, we have targeted the most general case.

In [23], the authors introduce a Choral library for handling protocols that might deliver messages out of order. Unlike Ozone, this library requires explicitly writing which parts of a choreography are independent. Dependencies between actions also need to be managed at a low level via side-effects. In our approach, out-of-order communications can be elegantly combined by using futures. Furthermore, the work in [23] does not deal with CIVs (which might arise if programmers are not careful) and presents no formal model.

A more loosely related line of research is that on multiparty session types (MPSTs) [19], where abstract choreographies without data or computation are used as protocol specifications that are compiled to "local session types". Similarly to most work on choreographic programming, some works on MPSTs allow for non-blocking send, but not non-blocking receive as in O_3. Previous work considered reordering actions in local session types [14], but these reorderings are necessarily limited because asynchronous multiparty session subtyping is undecidable in general [5]. Interprocedural MPSTs have been presented [15], but unlike O_3, the procedure calls require a central coordinator. Similar comments hold for recent investigations that add nondeterminism because of crashes [2, 32]. More generally, it is unclear whether "concurrency up to data dependency" could be expressed with MPSTs in their current form, since the types do not encode data dependencies.

7 Conclusion

We investigated a model for choreographic programming in which processes can execute out of order and messages can be reordered by the network. These features improve the performance of choreographies, without requiring programmers to rewrite their code, by allowing processes to better overlap communication with computation. However, compilers that use these features must have mechanisms in place to prevent communication integrity violations (CIVs). We presented a scheme to prevent CIVs by attaching dynamically-computed integrity keys to each message. Our results enlarge the class of behaviors that can be captured with choreographic programming without renouncing its correctness guarantees.

An important subject for future work is confluence. Statements can read and write to the local state of a process, so executing statements out of order can cause nondeterminism. Sometimes this nondeterminism is desirable (for instance, to implement consensus algorithms) but sometimes the nondeterminism is unexpected and causes bugs. In our formal model, nondeterminism could be controlled manually by allowing programmers to insert synthetic data dependencies. For example, below we use a hypothetical keyword $\mathsf{barrier_p}$ to prevent a file from being closed before it has been written-to:

$$\mathsf{val}\ \mathsf{p}.file = \mathit{open}(\text{“foo.txt”});\ \mathsf{p}.\mathit{write}(\mathsf{p}.file, \text{“hello”});\ \mathsf{barrier_p};\ \mathsf{p}.\mathit{close}(\mathsf{p}.file)$$

More generally, future work could develop a static analysis that identifies when two statements are not safe to execute out of order.

Another opportunity for static analysis to improve on our work concerns the size of session tokens. We chose to represent session tokens as lists of integers, which allowed processes to compute new session tokens without coordinating with one another. However, this encoding means the size of a token is proportional to the depth of the call stack—a problem for

tail-recursive programs such as *StreamIt* in Figure 7b. Fortunately, it is easy to see that communication integrity in *StreamIt* could be achieved in constant space by representing the token as a single integer, incremented upon each recursive call, assuming that processes do not participate in multiple instances of the choreography concurrently. With static analysis, a compiler could identify such programs and use a more efficient session token representation.

References

1. Akka. `https://akka.io/`, 2024.
2. Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2017. `doi:10.1007/978-3-319-60225-7\_1`.
3. Gul Agha. *ACTORS - a Model of Concurrent Computation in Distributed Systems*. MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1990.
4. Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGART Bulletin*, (64):55–59, August 1977. `doi:10.1145/872736.806932`.
5. Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: `https://lmcs.episciences.org/7238`.
6. Mario Bravetti, Ivan Lanese, and Gianluigi Zavattaro. Contract-driven implementation of choreographies. In Christos Kaklamanis and Flemming Nielson, editors, *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008. `doi:10.1007/978-3-642-00945-7\_1`.
7. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centered Programming for Web Services. *ACM Transactions on Programming Languages and Systems*, 34(2):1–78, June 2012. `doi:10.1145/2220365.2220367`.
8. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. `doi:10.1145/2429069.2429101`.
9. Luís Cruz-Filipe, Anne Madsen, Fabrizio Montesi, and Marco Peressotti. Modular choreographies: Bridging alice and bob notation to java. In Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmermann, editors, *Joint Post-proceedings of the Third and Fourth International Conference on Microservices, Microservices 2020/2022, May 10-12, 2022, Paris, France*, volume 111 of *OASIcs*, pages 3:1–3:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: `https://doi.org/10.4230/OASIcs.Microservices.2020-2022.3`, `doi:10.4230/OASICS.MICROSERVICES.2020-2022.3`.
10. Luís Cruz-Filipe and Fabrizio Montesi. On Asynchrony and Choreographies. *Electronic Proceedings in Theoretical Computer Science*, 261:76–90, November 2017. `doi:10.4204/EPTCS.261.8`.
11. Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017. `doi:10.1007/978-3-319-60225-7\_7`.

12 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Communications in choreographies, revisited. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1248–1255. ACM, 2018. doi:10.1145/3167132.3167267.

13 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. URL: https://doi.org/10.1007/s10817-023-09665-3, doi:10.1007/S10817-023-09665-3.

14 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 246–261. ACM, 2022. doi:10.1145/3503221.3508404.

15 Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi:10.1007/978-3-642-32940-1_20.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst.*, 46(1):1:1–1:59, 2024. doi:10.1145/3632398.

17 Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. Alice or bob?: Process polymorphism in choreographies. *J. Funct. Program.*, 34, 2024. URL: https://doi.org/10.1017/s0956796823000114, doi:10.1017/S0956796823000114.

18 Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

20 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies - computing preconditions in choreographic programming. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.

21 Shun Kashiwa, Gan Shen, Soroush Zare, and Lindsey Kuper. Portable, efficient, and practical library-level choreographic programming. *CoRR*, abs/2311.11472, 2023. URL: https://doi.org/10.48550/arXiv.2311.11472, arXiv:2311.11472, doi:10.48550/ARXIV.2311.11472.

22 Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In Antonio Cerone and Stefan Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 323–332. IEEE Computer Society, 2008. doi:10.1109/SEFM.2008.11.

23 Lovro Lugović and Fabrizio Montesi. Real-world choreographic programming: Full-duplex asynchrony and interoperability. *The Art, Science, and Engineering of Programming*, 8(2), October 2023. URL: http://dx.doi.org/10.22152/programming-journal.org/2024/8/8, doi:10.22152/programming-journal.org/2024/8/8.

24 Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, October 2004. doi:10.1017/S0960129504004323.

25 Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

26 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, Cambridge, 2023.

27 Johannes Aman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.27.

28 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

29 Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web*, pages 973–982, Banff Alberta Canada, May 2007. ACM. doi:10.1145/1242572.1242704.

30 Michael Scharf and Sebastian Kiesel. Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In *Proceedings of the Global Telecommunications Conference, 2006. GLOBECOM '06, San Francisco, CA, USA, 27 November - 1 December 2006*. IEEE, 2006. doi:10.1109/GLOCOM.2006.333.

31 Gan Shen, Shun Kashiwa, and Lindsey Kuper. Haschor: Functional choreographic programming for all (functional pearl). *Proc. ACM Program. Lang.*, 7(ICFP):541–565, 2023. doi:10.1145/3607849.

32 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.

33 Stephanie Wang, Eric Liang, Edward Oakes, Benjamin Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 671–686. USENIX Association, 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/cheng.

A Well-formedness

$$\mathsf{fv}(0) = \emptyset$$
$$\mathsf{fv}(\{\, C \,\};\, C') = \mathsf{fv}(C) \cup \mathsf{fv}(C')$$
$$\mathsf{fv}(l, t : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x;\, C') = \mathsf{fv}(e) \cup \mathsf{fv}(C') \setminus \{\mathsf{q}.x\}$$
$$\mathsf{fv}(l, t : \mathsf{val}\ \mathsf{p}.x = e;\, C') = \mathsf{fv}(e) \cup \mathsf{fv}(C') \setminus \{\mathsf{p}.x\}$$
$$\mathsf{fv}(l, t : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;\, C') = \mathsf{fv}(e) \cup \mathsf{fv}(C_1) \cup \mathsf{fv}(C_2) \cup \mathsf{fv}(C')$$
$$\mathsf{fv}(l, t : X(\overline{\mathsf{p}}, \overline{a});\, C') = \{\mathsf{p}.x \mid \mathsf{p}.x \in \overline{a}\} \cup \mathsf{fv}(C')$$
$$\mathsf{fv}(l, t : \overline{\mathsf{p}}.\, X(\overline{\mathsf{q}}, \overline{a})\, \{\, C \,\}) = \{\mathsf{p}.x \mid \mathsf{p}.x \in \overline{a}\} \cup \mathsf{fv}(C')$$
$$\mathsf{fv}(I;\, C) = \mathsf{fv}(C) \text{ otherwise.}$$

$$\frac{\begin{array}{c}\mathsf{pn}(C) \subseteq \mathsf{dom}(\Sigma) \qquad \mathsf{pn}(C) \subseteq \mathsf{dom}(K) \qquad \mathsf{fv}(C) = \emptyset \\ \mathsf{keys}(C)\ \text{distinct} \qquad \forall (l, t) \in \mathsf{keys}(C),\, t \neq \mathsf{t} \\ \forall I_1, I_2 \in \mathsf{stats}(C),\ \text{if}\ \mathsf{key}(I_1) \prec \mathsf{key}(I_2)\ \text{then}\ I_1 = l_1, t_1 : \overline{\mathsf{q}}.X(\overline{\mathsf{p}}, \overline{a})\{C'\}\ \text{and}\ I_2 \in \mathsf{stats}(C') \\ (X(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C)\checkmark\ \text{for each}\ (X(\overline{\mathsf{p}}, \overline{\mathsf{p}.x}) = C) \in \mathscr{C} \qquad \langle I, K\rangle \checkmark\ \text{for each}\ I \in \mathsf{stats}(C)\end{array}}{\langle \mathscr{C}, C, \Sigma, K\rangle \checkmark}\ \text{C-WF}$$

$$\frac{\exists! v,\ (l, \tau, v) \in K(\mathsf{q})}{\langle l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x,\ K\rangle \checkmark}\ \text{C-WF-Recv}$$

$$\frac{(l, \tau, L) \in K(\mathsf{q})}{\langle l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}[L],\ K\rangle \checkmark}\ \text{C-WF-OnSelect}$$

$$\frac{}{\langle l, \tau : \mathsf{val}\ \mathsf{p}.x = e,\ K\rangle \checkmark}\ \text{C-WF-Compute}$$

$$\frac{C_1, C_2\ \text{contain no runtime terms}}{\langle l, \tau : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2,\ K\rangle \checkmark}\ \text{C-WF-If}$$

$$\frac{}{\langle \{\, C_1 \,\},\ K\rangle \checkmark}\ \text{C-WF-Block}$$

$$\frac{\begin{array}{c}(X(\mathsf{q}_1, \ldots, \mathsf{q}_n, \mathsf{q}^1.x_1, \ldots, q^m.x_m) = C) \in \mathscr{C} \\ \mathsf{p}_1, \ldots, \mathsf{p}_n\ \text{distinct} \qquad \forall i \leq n, j \leq m,\ \text{if}\ \mathsf{pn}(a_j) = \mathsf{p}_i\ \text{then}\ \mathsf{q}^j = \mathsf{q}_i\end{array}}{\langle l, \tau : X(\mathsf{p}_1, \ldots, \mathsf{p}_n, a_1, \ldots, a_m),\ K\rangle \checkmark}\ \text{C-WF-Call}$$

B EPP Theorem

Proving the EPP Theorem requires first establishing several basic properties about choreographies. Lemmas 7 and 8 are properties of substitution: Lemma 7 states that local variables at a process p are projected to local variables at that same location in the network and Lemma 8 states that substitution preserves the $(\sqsupseteq)$ relation. Lemma 9 states that projection preserves sequential composition—a modified version of Lemma 8.17 from Montesi [26].

▶ **Lemma 7.**

$$[\![C[\mathsf{p}.x \mapsto v]]\!]_{\mathsf{r}} = \begin{cases} [\![C]\!]_{\mathsf{p}}[x \mapsto v] & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C[\mathsf{p}.x \mapsto v]]\!]_{\mathsf{q}} = [\![C]\!]_{\mathsf{q}} & \text{otherwise.} \end{cases}$$

▶ **Lemma 8.** *$P \sqsupseteq Q$ implies $P[x \mapsto v] \sqsupseteq Q[x \mapsto v]$.*

▶ **Lemma 9.** *$[\![I; C]\!]_{\mathsf{q}} = [\![I; 0]\!]_{\mathsf{q}} \fatsemi [\![C]\!]_{\mathsf{q}}$ if I does not have the form $l, t : \mathsf{p} \to \mathsf{q}[L]$ or $l, t : \mathsf{p} \rightsquigarrow \mathsf{q}[L]$.*

We can now prove the EPP Theorem, which follows from standard proof techniques. The only novelty is (1) our use of substitution instead of local state to bind variables to values; (2) our use of well-formedness; and (3) a large number of extra cases in the soundness direction, generated by the P-Delay rule. Fortunately, all these extra cases are dispatched easily. We include a proof sketch below to show how this is done.

▶ **Theorem 6** (EPP Theorem). *Let $\langle C,\ \Sigma,\ K \rangle$ be a well-formed configuration.*

1. *(Completeness) If $\langle C,\ \Sigma,\ K \rangle \xrightarrow{\mathsf{p}} \langle C',\ \Sigma',\ K' \rangle$ then $\langle [\![C]\!],\ \Sigma,\ K \rangle \xrightarrow{\mathsf{p}} \langle N',\ \Sigma',\ K' \rangle$ for some well-formed N' where $N' \sqsupseteq [\![C']\!]$.*
2. *(Soundness) If $\langle N,\ \Sigma,\ K \rangle \xrightarrow{\mathsf{r}} \langle N',\ \Sigma',\ K' \rangle$ for some well-formed N where $N \sqsupseteq [\![C]\!]$, then $\langle C,\ \Sigma,\ K \rangle \xrightarrow{\mathsf{p}} \langle C',\ \Sigma',\ K' \rangle$ for some C' where $N' \sqsupseteq [\![C']\!]$.*

Completeness, sketch. The proof proceeds by induction on the derivation $\mathcal{D}$ that produces $\langle C,\ \Sigma,\ K \rangle \xrightarrow{\mathsf{p}} \langle C',\ \Sigma',\ K' \rangle$. In most cases it suffices to let $N' = [\![C']\!]$. In the case of C-If, we find a network N' such that $N' \neq [\![C']\!]$ but $N' \sqsupseteq [\![C']\!]$.

Because O_3 uses scoped variables, the proof requires Lemma 7. The C-Delay rule is also slightly novel: if I is not a selection at q then completeness requires Lemma 9 and an application of P-Delay. ◀

Soundness, sketch. The proof proceeds by induction on the structure of the choreography C. The base case, $C \equiv 0$, is trivial. Otherwise, $C \equiv (I; C')$ and there is a distinct case for each instruction I. The key novelty in this proof is handling P-Delay and the frequent use of well-formedness to guarantee that K does not contain certain messages. We consider two representative cases.

Case 1. Let $C = l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x;\ C''$. Then, by the definition of EPP, N has the form

$$N = \mathsf{p}[\mathsf{q}\ !_{l,\tau}\ e;\ P] \mid \mathsf{q}[?_{l,\tau}\ x;\ Q] \mid (N \setminus \mathsf{p}, \mathsf{q}).$$

There are three sub-cases.

Case 1.1. Assume $\mathsf{r} = \mathsf{p}$. In standard choreography models, this case can only proceed by P-Send. In our model, it could also proceed by P-Delay. Hence there are two sub-cases:
Case 1.1.1. (P-Send) Satisfied by letting $C' = l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x;\ C''$.
Case 1.1.2. (P-Delay) By the induction hypothesis, there exists C''' such that $N''' \sqsupseteq [\![C''']\!]$. The case is then satisfied by letting $C' = l, \tau : \mathsf{p}.e \to \mathsf{val}\ \mathsf{q}.x;\ C'''$.

Case 1.2. Assume $\mathsf{r} = \mathsf{q}$. Again, we consider the two rules by which the case could proceed:
Case 1.2.1. (P-Recv) We must show that it is impossible for q to receive a message in $\langle N,\ \Sigma,\ K \rangle$. Since C is well-formed, it cannot contain a communication-in-progress term $l, \tau : \mathsf{p} \rightsquigarrow \mathsf{q}.x$ with the integrity key (l, τ). Hence M does not contain any messages of the form (l, τ, v).

Case 1.2.2. (P-Delay) This case proceeds similarly to the previous P-Delay case.

Case 1.3. Assume $\mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\}$. Follows from the induction hypothesis, as in Case 1.1.2.

Case 2. Let $C = l, \tau : \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2;\ C_3$. Then N has the form

$$N = \mathsf{p}[\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2;\ P_3] \mid \left(\prod_{q_i \in \overline{\mathsf{q}}} \mathsf{q}_i[\&\{(l_j, \tau_j, L_j) \Rightarrow Q_{i,j}\}_{j \in \mathcal{I}};\ Q_i] \right) \mid (N \setminus \mathsf{p}, \overline{\mathsf{q}}).$$

Consider the case where $\mathsf{r} = \mathsf{q}_i \in \overline{\mathsf{q}}$ and $\langle\, N,\ \Sigma,\ K \,\rangle \xrightarrow{\mathsf{q}} \langle\, N',\ \Sigma',\ K' \,\rangle$ proceeds by P-OnSelect. That is,

$$\mathsf{q}_i[\&\{(l_j, \tau_j, L_j) \Rightarrow Q_{i,j}\}_{j \in \mathcal{I}};\ Q_i] \xrightarrow{\mathsf{q}_i} \mathsf{q}_i[Q_{i,k};\ Q_i]$$

for some $k \in \mathcal{J}$. We must show that this case is impossible. Notice that the step can only occur if $(l, \tau_k, L_k) \in K(\mathsf{q}_i)$. Such a message can only occur if $l, \tau_k : \mathsf{p} \rightsquigarrow \mathsf{q}[L_k]$ occurs in C_1, C_2, or C_3. Because C is well-formed, C_1 and C_2 do not contain runtime terms; hence the term could only occur in C_3. But then Q_i would contain a branch $(l, \tau_k, L_k) \Rightarrow Q'$ and $\mathsf{keys}(N|_{\mathsf{q}_i})$ would not be distinct; a contradiction of Lemma 5 (5). ◀