


Modular Choreographies: Bridging Alice and Bob Notation to Java

Luís Cruz-Filipe ✉ 

University of Southern Denmark, Odense, Denmark

Anne Madsen

University of Southern Denmark, Odense, Denmark

Fabrizio Montesi ✉ 

University of Southern Denmark, Odense, Denmark

Marco Peressotti ✉ 

University of Southern Denmark, Odense, Denmark

Abstract

We present Modular Choreographies, a new choreographic programming language that features modular functions. Modular Choreographies is aimed at simplicity: its communication abstraction follows the simple tradition from the “Alice and Bob” notation. We develop a compiler toolchain that translates choreographies into modular Java libraries, which developers can use to participate correctly in choreographies. The key novelty is to compile through the Choral language, which was previously proposed to define object-oriented choreographies: our toolchain compiles Modular Choreographies to Choral, and then leverages the existing Choral compiler to generate Java code. Our work is the first to bridge the simplicity of traditional choreographic programming languages with the requirement of generating modular libraries in a mainstream language (Java).

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed programming languages; Applied computing → Service-oriented architectures

Keywords and phrases Choreographic Programming, Choreographies, Modularity

Digital Object Identifier 10.4230/OASICS.Microservices.2020-2022.3

Supplementary Material *Software*: <https://github.com/chorlang/modular-choreographies>
archived at `swh:1:dir:7550e65d0b3b2fbcf59e1e2e52eef42b79ae12c5`

Funding *Fabrizio Montesi*: Work partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

1 Introduction

A recognised best practice for the development of microservices [19] is to coordinate them according to *choreographies*: coordination plans that prescribe how processes in a distributed system should interact with each other, by exchanging messages [32]. However, writing programs that comply with a choreography falls under the shadow of writing correct concurrent and distributed software, which is notoriously hard even for experts [26]. This is due to the well-known state explosion problem: even for small programs, the number of possible ways in which they could interact can grow exponentially and reach unmanageable numbers [6, 35].

Choreographic programming is a programming paradigm where programs are choreographies [31]. Its aim is to relieve programmers from implementing choreographies manually, by following two steps: first, programmers can code the choreography that they wish for by using a programming language equipped with primitives that make interactions syntactically manifest; then, a compiler automatically generates a working implementation of the choreography. The theory of choreographic programming has been explored in several directions, including service-oriented computing [4], adaptability [17], cyber-physical systems [28], functional correctness [25], and security [27, 2].



© Luís Cruz-Filipe, Anne Madsen, Fabrizio Montesi, and Marco Peressotti;
licensed under Creative Commons License CC-BY 4.0

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).

Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 3; pp. 3:1–3:18



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3:2 Modular Choreographies

Choreographic programming languages are inspired by security protocol notation (also known as “Alice and Bob” notation), which was introduced for the definition of security protocols [33]. The key primitive of these languages is the interaction term $A.\text{expr} \rightarrow B.x$ which reads “A communicates the result of expression expr to B, which stores it in its local variable x ”. The participants A and B are called processes, or roles [12, 3].

Until recently, implementations of choreographic programming languages mainly generated standalone systems and did not provide means to integrate the output code with mainstream development practices [31, 4, 17]. The Choral programming language was later proposed as the first choreographic programming language that can be applied to mainstream programming [20]. In Choral, a choreography is compiled to a Java library for each process described in the choreography. A developer can then import this library and invoke it to play the part of that process in a distributed system.

To achieve Java interoperability, choreographies in Choral are less abstract than usual. Developers have to take care of how communications are supported by concrete communication channels, how data types can be expressed in Java, and how choreographic functions should be structured in terms of classes and methods. Also, the simple interaction term $A.\text{expr} \rightarrow B.x$ has to be written as a method invocation instead, like the following.

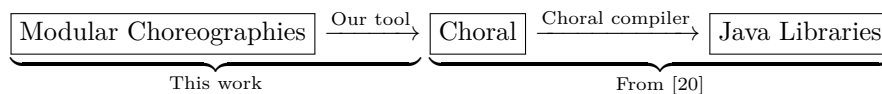
```
var x@B = channel.com( MyClass@A.expr() );
```

The previous line of code reads “variable x at B is assigned the result of invoking the static method expr of MyClass at A and passing it through an invocation of method com of object channel (which moves data from A to B)”. Understanding Choral code thus requires more knowledge. While these aspects are essential for bridging choreographies to real-world Java programs, they force designers to mix choreographies with implementation details that diminish their level of abstraction, and thus hinder reusability of choreographies for different settings.

In this paper, we bridge the gap between the traditional simplicity of choreographic languages with the practicality of Choral. Specifically, we present the following contributions:

- A new choreographic programming language, called Modular Choreographies, and its formal semantics. Modular Choreographies offers a simple choreographic syntax with the standard “Alice and Bob” communication primitive ($A.\text{expr} \rightarrow B.x$), augmented with linguistic constructs for writing parametric functions. Our design is purposefully inspired by previous choreographic languages to be familiar (more details are given in Section 2).
- A type system for Modular Choreographies, which checks that functions are invoked correctly: the processes that enact a function have access to the right data and local functions (e.g., for encryption). Our type system supports the expected property of subject reduction [37].
- An implementation of Modular Choreographies, consisting of a parser and a type checker.
- A tool that, given code in Modular Choreographies, synthesises a program in Choral. The synthesiser automatically generates classes, methods, and the necessary usages of channels in order to move data correctly as instructed by the choreography.

Taken together, our contributions and Choral enable a new development methodology for implementing software that follows choreographies correctly, which we depict below.



That is, developers can use Modular Choreographies to design protocols expressed in a simple choreographic language and generate valid Choral code (using our tool), from which compliant Java libraries can be automatically generated (using the compiler from [20]). This gives choreography designers the option of using a simple language, without giving up on Java interoperability. If the decisions made by our tool when synthesising the Choral code need to be refined, it can be done before the Java libraries are generated; for example, it is possible to change method names or the type used to denote data that can be transmitted (the default is `Serializable`). This is enabled by our two-step approach and, we believe, is better than editing the Java libraries directly: once we reach that level, code does not have a choreographic view anymore and therefore introducing concurrency bugs is much easier.

Using Choral as intermediate technology also gives the pragmatic advantage of reusing what already exists to a reasonable extent. In particular, we do not implement yet another procedure for compiling choreographies to separate distributed programs – a process known as Endpoint Projection [3]. This allowed us to focus on the design of Modular Choreographies and the novel aspect of connecting “Alice and Bob” choreographies to object orientation.

Structure of the Paper. Section 2 discusses relevant related work. Modular Choreographies and its type system are presented in Section 3. We recap useful background knowledge on Choral and illustrate how our implementation works in Section 4.¹ Conclusions and future work are given in Section 5.

2 Related Work

We discuss here the most related work and highlight important differences. The reader interested in an introduction to choreographic languages and their compilation can consult [32].

The design of Modular Choreographies is inspired by the theories of Procedural Choreographies [11] and Recursive Choreographies [32]. Notably, it inherits the features that processes (can) have mutable states, choreographies are parameterised over the processes that enact them, and function calls can have continuations (general recursion). Differently from these theories, Modular Choreographies includes linguistic constructs for defining and using functions that contain choreographies more modularly, motivated by the intention of using our language for practical programming. Relevant changes include: functions can have local variables; choreographies can be parameterised over the local functions that processes can run; the capability of returning values from a function, possibly at many different processes; and a type system for checking that procedures are invoked by passing arguments of the right types. We deal with the combination of mutable state, value return, and general recursion by introducing call frames to the operational semantics of choreographies. Procedures with local scopes were already present in the first choreographic programming language, Chor [4, 31], but they could not be parameterised over processes nor local functions. Other implemented choreographic programming languages that predate our work but do not offer modularity include AIOCJ [17] and hacc [9]. One choreographic programming language that does support modularity is HasChor, an embedded domain-specific language in Haskell [40]. Differently from our toolchain (and Choral, which we leverage), HasChor requires each participant to know the entire choreography in order to be executed, whereas we compile separate individual libraries; also, it injects broadcast communications for coordinating choices that

¹ At the time of this writing, the latest version of our implementation is available at <https://github.com/chorlang/modular-choreographies>.

are not always necessary in our case, while we use Choral’s implementation of the “merging” operator [3] to detect that choices are communicated correctly among participants (a property known as “knowledge of choice” [5, 32]).

Our development is also based on Choral, which introduced the first integration between choreographic languages and mainstream programming abstractions [20]. In particular, Choral’s types give control over the APIs that are generated in the target Java libraries. We use this control extensively to compile APIs that allow programmers to pass parameters to our Modular Choreographies (e.g., functions and data locally available at processes) and to use the values that choreographies return. Choral is also the first choreographic programming language that introduced returning values and higher-order choreographies (choreographies parameterised over choreographies), but unlike Modular Choreographies it does not offer a syntax based on the “Alice and Bob” notation and it does not come with a formal semantics.

Previous work studied theories of choreographic languages based on the “Alice and Bob” notation for several settings, including verification [8, 25, 7], cyberphysical systems [29, 28], security [27, 2], web services [3], and transactions [41]. Modular Choreographies has the potential of being an interesting basis for the future application of these theories.

Another paradigm related to choreographic programming is multitier programming, which shares the practice of compiling a program into the distributed implementations of multiple participants [43]. Differently from choreographic programming, multitier programs are not choreographies. Multitier code describes the local computation that a participant performs from its own viewpoint. However, this viewpoint can be switched to that of another participant by performing a “hop” that changes the scope of variables. For example, a function declared to be located at the client can seamlessly invoke a function declared to be located at the server. The program would then be split by a compiler to code for the client and server, and the hop would typically be implemented at runtime by a coordination middleware. The control that choreographies give over how participants communicate is known to be important for writing faithful implementations of protocols [30]. Nevertheless, at least for some multitier languages, it is possible to translate a multitier program into a choreography that implements it, which basically boils down to synthesising a precise interaction protocol that runs the multitier program [21].

There are other tools for the generation of libraries that aid developers with following communication flows, based on “global types”: abstractions of choreographies that specify only the types of data to be communicated [23]. For example, in [39] global types are compiled into local specifications for the communication behaviour that each participant should implement, which are then used to generate libraries with fluid APIs that aid developers with ordering send/receive actions correctly (like `o.send(...).receive(...).send(...)`). The programmer must then implement these actions by hand. This is a less direct approach than choreographic programming, where the programmer just needs to define the source choreography and then correct implementations are automatically generated.

3 Modular Choreographies

Syntax. The syntax of Modular Choreographies is given by the grammar in Figure 1. We denote lists of similar elements with overlines e.g., writing \bar{e} for e_1, \dots, e_n , and optionally indicating a index variables e.g., \bar{e}_i . Processes are identified by process names (p, q, \dots), they execute concurrently, are equipped with a local (private) memory that supports dynamic allocation, and can evaluate (local) expressions (e). We fix a minimal syntax for local expressions: v ranges over value literals, id ranges over identifiers for variables and (local)

$e ::= c \mid id(\bar{e})$	Local expressions
$c ::= v \mid id$	
$b ::= \text{int} \mid \text{string} \mid \text{boolean} \mid \text{double} \mid \dots$	Local data types
$t ::= b \mid \bar{b} \rightarrow b$	Local types
$C ::= \varepsilon \mid I;C$	Choreographies
$I ::= \mathbf{var} \ p.id:b$	Variable declaration
$\mid p.id = e$	Variable assignment
$\mid \mathbf{var} \ p.id:b = e$	
$\mid p.e \rightarrow q.id$	Value communication
$\mid p.e \rightarrow \mathbf{var} \ q.id:b$	
$\mid p \rightarrow q[l]$	Label selection
$\mid \mathbf{if} \ p.e \{C\} \ \mathbf{else} \ \{C\}$	Conditional
$\mid \overline{p.(id)} = ID(\overline{p.(\bar{e})})$	Function call
$\mid \mathbf{return} \ \overline{p.(\bar{e})}$	Function return
$C ::= \left\{ ID(p:\mathbf{proc}(\bar{id}:t)) : (\overline{p:\mathbf{proc}(\bar{b})}) \{C\} \right\}_{i \in I}$	Function definitions

■ **Figure 1** Modular Choreographies, syntax (syntactic sugar is greyed).

functions, and $id(\bar{e})$ denotes application. We denote the set of value literals for a local data type $\text{Values}(b)$. Choreographies (C) are sequences of choreographic instructions I with ε denoting an empty sequence. In the instruction $\mathbf{var} \ p.id:b$, p declares a variable id with type b ; and in $p.id = e$ it assigns to id of the result of evaluating the expression e ; $\mathbf{var} \ p.id:b = e$, p performs both a declaration and an assignment. In $p.e \rightarrow q.id$, p communicates the result of evaluating the local expression e to q which stores it in its local variable id and in $p.e \rightarrow \mathbf{var} \ q.id:b$ the q also declares the variable id . In $p \rightarrow q[l]$, p communicates label l (a constant distinct from values used by local computation) to q ; this type of communication does not alter the state of either process and is used to propagate decisions about control flow (as common for choreographic languages). In $\mathbf{if} \ p.e \{C\} \ \mathbf{else} \ \{C\}$, p evaluates the (boolean) guard e and the choreography proceeds with either branch accordingly. In $\overline{p.(id)} = ID(\overline{p.(\bar{e})})$, \bar{p} call the choreographic function ID on the values obtained evaluating the formal arguments $\overline{p.(\bar{e})}$ (each argument is at a single process) and assign the result to the variables $\overline{p.(id)}$. Finally, in $\mathbf{return} \ \overline{p.(\bar{e})}$, each process participating in a function call returns to the caller with the values obtained by evaluating its local expressions as result. Note that choreographic functions may return multiple values at multiple processes, e.g., a function for establishing a shared secret among two parties will return a value at each of them as illustrated in the next example.

► **Example 1.** The Diffie-Hellman key-exchange protocol [18] allows two parties **a** and **b** to establish a shared secret key for symmetric encryption. The protocol assumes that the two participants share a prime number m and a primitive root modulo m , g , that each has a private key `privKey`, and that both can perform modular exponentiation `exp`. To implement this protocol we define a choreographic function `DH` that, for each participant, takes the arguments `privKey:int, g:int, m:int, exp:(int,int,int)->int` and returns a value of type `int`.

3:6 Modular Choreographies

```
// Diffie-Hellman key-exchange protocol
DH(a:proc(privKey:int,g:int,m:int,exp:(int,int)->int),
  b:proc(privKey:int,g:int,m:int,exp:(int,int)->int)
):(a:proc(int),b:proc(int)) {
  a.exp(privKey,g,m) -> var b.pubKey:int; // a computes and sends its public key to b
  b.exp(privKey,g,m) -> var a.pubKey:int; // b computes and sends its public key to a
  var a.key:int = exp(privKey,pubKey,m); // a computes its copy of the shared secret
  var b.key:int = exp(privKey,pubKey,m); // b computes its copy of the shared secret
  return a.key, b.key; // a and b return the shared secret
}
```

MC Code

► **Example 2.** In this example we provide an implementation in MC of the Single Sign On protocol defined in [32] using Recursive Choreographies. In this protocol a client *c* gains access to a service *s* by getting its credentials checked by a third party credential authentication service *a*. If the check succeeds the service *s* will issue a token to *c* otherwise they will proceed with another attempt.

```
SSO(c:proc(creds:()->int),
  s:proc(newToken:()->int),
  a:proc(valid:(int)->boolean)
):(c:proc(int)) {
  c.creds() -> var a.x:int // the client sends credentials to the authority
  var c.t:int
  if a.valid(x) // the authority checks the credentials it received
  {
    a -> s[OK] // and informs the service it guarantees for the client
    s -> c[TOKEN] // which informs the client it will receive a token
    s.newToken() -> c.t // and issues a token
  } else {
    a -> s[KO]
    s -> c[ERROR]
    c.t = SSO(c.creds,s.newToken,a.valid) // retry
  }
  return c.t // return the token at the client
}
```

MC Code

Semantics. We specify the expected behaviour of Modular Choreographies by means of an operational semantics which we will later use to validate the design of the MC type system. Our design follows the same principles used by the models we extend ([12, 11]) but dispenses from out-of-order execution of actions at distinct processes (e.g., in $p.id = e; q.id = e$, the processes p and q can, in practice, carry out their local computations independently).² The semantics can be readily extended to account for this aspect following the same approach based on delay actions used by *loc. cit.* and minor changes to our proofs. However, the return on this investment in complexity of the model is limited and does not result in stronger results for the aims of this work since Choral (the compilation target for MC) lacks a formal semantics.

² This simplification is not a first in the literature of choreographic programming: [13] defines both sequential and concurrent semantics for its choreographic language arguing that the former is precise enough to support reasoning about program correctness; [12, 14] shows that for establishing many results of interest it is sufficient to consider “head transitions” which correspond to establishing a sequential semantics.

$$\begin{array}{c}
\boxed{\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma', R', \Sigma' \rangle} \\
\hline
\frac{p.id \notin \Gamma}{\langle \Gamma, \mathbf{var} \ p.id : b ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, p.id : b, C, \Sigma[p.id := \text{default}(b)] \rangle} \text{SDEC} \\
\frac{\Sigma(p) \vdash e \downarrow v}{\langle \Gamma, p.id = e ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, C, \Sigma[p.id := v] \rangle} \text{SAss} \\
\frac{\Sigma(p) \vdash e \downarrow v}{\langle \Gamma, p.e \rightarrow q.id ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, C, \Sigma[q.id := v] \rangle} \text{SCom} \\
\frac{}{\langle \Gamma, p \rightarrow q[l] ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, C, \Sigma \rangle} \text{SSEL} \\
\frac{\Sigma(p) \vdash e \downarrow v}{\langle \Gamma, \mathbf{if} \ p.e \{C_{\text{true}}\} \ \mathbf{else} \ \{C_{\text{false}}\} ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, C_v \ ; C, \Sigma \rangle} \text{SCond} \\
\frac{\overline{\Sigma(p_k) \vdash e_{k,l} \downarrow c_{k,l}} \quad \overline{ID(q_k : \mathbf{proc}(\overline{id_{k,l} : t_{k,l}})) : (q : \mathbf{proc}(\overline{b})) \{C'\} \in \mathcal{C}}}{\langle \Gamma, p.(\overline{id}) = ID(\overline{p_k.(\overline{e_{k,l}})}) ; C, \Sigma \rangle \longrightarrow_C} \text{SCall} \\
\frac{\langle \Gamma, r_i.(\overline{id_{i,j}}) = \overline{p_k.id_{k,l} : t_{k,l}, C'[p_k/q_k], \Sigma[p_k.id_{k,l} := c_{k,l}]} ; C, \Sigma \rangle}{\langle \Gamma', R', \Sigma' \rangle \longrightarrow_C \langle \Gamma', R', \Sigma' \rangle} \text{SCallRT} \\
\frac{\langle \Gamma, p.(\overline{id}) = \langle \Gamma', R', \Sigma' \rangle ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, r.(\overline{id}) = \langle \Gamma', R', \Sigma' \rangle ; C, \Sigma \rangle}{\overline{\Sigma'(p_i) \vdash e_{i,j} \downarrow v_{i,j}}} \text{SCallRET} \\
\langle \Gamma, p_i.(\overline{id_{i,j}}) = \langle \Gamma', \mathbf{return} \ \overline{p_i.(\overline{e_{i,j}})}, \Sigma' \rangle ; C, \Sigma \rangle \longrightarrow_C \langle \Gamma, C, \Sigma[p_i.id_{i,j} := v_{i,j}] \rangle
\end{array}$$

■ **Figure 2** Modular Choreographies, semantics.

The semantics of MC is given as a transition whose states are triples $\langle \Gamma, R, \Sigma \rangle$ where:

- Γ tracks the variables and functions available at each process together with their type.
- R is a choreography term (in the syntax of MC extended with runtime terms that we will introduce below).
- Σ tracks the memory state of each process.

We represent typing environments using the grammar below assuming processes occur at most once in Γ and identifiers occur at most once in each γ .

$$\Gamma ::= \cdot \mid \Gamma, p.\gamma \quad \gamma ::= \cdot \mid \gamma, id : t$$

We refer to Γ and γ as global and local, respectively and write $\Gamma(p)$ for the environment γ local to p in Γ (given that p occurs in Γ). We extend the syntax of MC with a runtime term for representing the call stack:

$$R ::= C \mid \overline{p.(\overline{id})} = \langle \Gamma, R, \Sigma \rangle ; C$$

In the $\overline{p.(\overline{id})} = \langle \Gamma, R, \Sigma \rangle ; C$, R is executed under Γ and Σ and the result of this execution will be stored under $p.(\overline{id})$ before continuing with C . (This design avoids the need to define frames for Γ and Σ .) The memory state Σ can be regarded as a mapping assigning each process p to its local memory state (denoted as $\Sigma(p)$). We write $\Sigma[p.id := v]$ for the state obtained by assigning v to $p.id$ in Σ . We assume that each basic type b comes with a default value and denote this value by $\text{default}(b)$.

3:8 Modular Choreographies

To evaluate local expressions we assume an evaluation function that takes a local state as a parameter. We write $\Sigma(p) \vdash e \downarrow c$ to denote that the local expression e evaluates to c under the state $\Sigma(p)$ (local at p).

Transitions are specified by the derivation rules reported in Figure 2. Rules SDEC–SSEL capture the intuition behind the different choreographic primitives given earlier. Rule SCOND models the behaviour of a conditional where p chooses which choreography to execute based on the outcome of evaluating its guard (locally). In the transition target, $C_v \mathbin{;} C$ denotes the choreography obtained by “grafting” C onto each continuation point (ε not guarded by a return) in C_v according to the recursive definition below.

$$C \mathbin{;} C' = \begin{cases} C' & \text{if } C = \varepsilon \\ C & \text{if } C = \mathbf{return} \overline{p.(\bar{e})}; \varepsilon \\ I; (C \mathbin{;} C') & \text{otherwise} \end{cases}$$

Rules SCALL–SCALLRET model the initialisation, execution, and termination of a function call. Rule SCALL creates a fresh frame for the execution of function ID where the environment and choreography are given by the formal parameters of the function and its body where all processes (\mathbf{q}_k) are instantiated with those provided by the call site (\mathbf{p}_k), and the memory state is initialised with the actual parameters ($c_{k,l}$). Observe that actual parameters can be basic values (v) or identifiers (id) since functions in MC can be parametrised in the names of local functions. Rule SCALLRT allows for the execution of a choreography in a call frame. Rule SCALLRET models a call returning a result by removing the corresponding frame and storing the result in the state of the caller.

Typing and progress. We equip MC with a typing discipline that checks that variables, values, and functions (local or choreographic) are used according to their declared type and that results returned by choreographic functions are of the expected type. The typing judgments and derivation rules that compose the type system are summarised in Figure 3.

The typing discipline uses the local types b and t and signatures of choreographic functions found in the syntax of MC. Additionally, it uses types given by the following grammar to express whether every, some, or none of the exit points (ε and $\mathbf{return} \overline{p.(\bar{e})}$) of a choreography return values of a given type.

$$S ::= \{B\} \mid \{\perp\} \mid \{\perp, B\} \quad B ::= \overline{p:\mathbf{proc}(b)}$$

A type of the first form describes a choreography where every exit point returns values according to B , a type of the second describes a choreography where every exit point is without a return instruction, and the third describes a choreography with exit point for both cases (returning values of type B or no return at all). This information will be important for choreographies with conditionals where only few branches return. To combine information about exit points in different branches of a conditional we introduce a “join” operation $S_1 \curlywedge S_2$ which is defined as $S_1 \cup S_2$ wherever S_1 and S_2 agree on the type of returned values in the sense that $B_1 \in S_1, B_2 \in S_2 \implies B_1 = B_2$. For instance, $\{\perp\} \curlywedge \{B\}$ and $\{\perp\} \curlywedge \{\perp, B\}$ both yield $\{\perp, B\}$ whereas $\{p.\mathbf{int}\} \curlywedge \{p.\mathbf{boolean}\}$ is undefined.

Additionally to global (Γ) and local (γ) typing environments, the typing discipline relies on separate typing environments (Δ) for recording the choreographic functions available and their signature. These environments are represented using the grammar below under the assumption that each ID occurs at most once.

$$\Delta ::= \cdot \mid \Delta, ID: \overline{q_i:\mathbf{proc}(t_{i,k})} \rightarrow \overline{q_i:\mathbf{proc}(b_{i,j})}$$

$\gamma \vdash e : t$	$\frac{v \in \text{Values}(b)}{\gamma \vdash v : b} \text{TLIT} \quad \frac{id : \bar{b} \rightarrow b \in \gamma}{\gamma \vdash id : \bar{b} \rightarrow b} \text{TID} \quad \frac{\gamma \vdash id : b_1, \dots, b_n \rightarrow b \quad \overline{\gamma \vdash e_i : b_i}}{\gamma \vdash id(e_1, \dots, e_n) : b} \text{TAPP}$
$\Gamma \vdash \Sigma$	$\frac{}{\cdot \vdash \Sigma} \quad \frac{\Gamma(p) \vdash \Sigma(p)(id) : t \quad \Gamma \vdash \Sigma}{\Gamma, p. id : t \vdash \Sigma}$
$\Delta; \Gamma \vdash R : S$	$\frac{}{\Delta; \Gamma \vdash \varepsilon : \{\perp\}} \text{TNIL} \quad \frac{\overline{\Gamma(p_i) \vdash e_{i,j} : b_{i,j}}}{\Delta; \Gamma \vdash \mathbf{return} \overline{p_i.(e_{i,j})}; \{p_i : \mathbf{proc}(b_{i,j})\}} \text{TRET}$ $\frac{id \notin \gamma \quad \Delta; \Gamma, p.(\gamma, id : b) \vdash C : S}{\Delta; \Gamma, p. \gamma \vdash \mathbf{var} \overline{p. id : b}; C : S} \text{TDEC} \quad \frac{\Gamma(p) \vdash e : b \quad \Gamma(q) \vdash id : b \quad \Delta; \Gamma \vdash C : S}{\Delta; \Gamma \vdash p.e \rightarrow q.id; C : S} \text{TCOM}$ $\frac{\Gamma(p) \vdash id : b \quad \Gamma(p) \vdash e : b \quad \Delta; \Gamma : b \vdash C : S}{\Delta; \Gamma \vdash p.id = e; C : S} \text{TASS} \quad \frac{p, q \in \Gamma \quad \Delta; \Gamma \vdash C : S}{\Delta; \Gamma \vdash p \rightarrow q[l]; C : S} \text{TSEL}$ $\frac{\Gamma(p) \vdash e : \text{boolean} \quad \Delta; \Gamma \vdash C_i : S_i \quad \perp \notin (S_1 \vee S_2) \implies C_3 = \varepsilon}{\Delta; \Gamma \vdash \mathbf{if} \overline{p.e \{C_1\}} \mathbf{else} \{C_2\}; C_3 : S_1 \vee S_2 \vee S_3} \text{TCOND}$ $\frac{\overline{\Gamma \vdash p_k.e_{i,k} : t_{i,k}} \quad \overline{\Gamma \vdash p_i.id_{i,j} : b_{i,j}} \quad \Delta; \Gamma \vdash C : S}{\Delta, ID : \overline{q_i : \mathbf{proc}(t_{i,k})} \rightarrow \overline{p_i : \mathbf{proc}(b_{i,j})}; \Gamma \vdash \overline{p_i.(id_{i,j})} = ID(\overline{p_i.(e_{i,k})}); C : S} \text{TCALL}$ $\frac{\Delta; \Gamma' \vdash R : \{p_i : \mathbf{proc}(b_{i,j})\} \quad \Gamma' \vdash \Sigma' \quad \overline{\Gamma \vdash p_i.id_{i,j} : b_{i,j}} \quad \Delta; \Gamma \vdash C : S}{\Delta; \Gamma \vdash \overline{p_i.(id_{i,j})} = (\Gamma', R, \Sigma); C : S} \text{TCALLRT}$
$\Delta \vdash C$	$\frac{ID : \overline{p_i : \mathbf{proc}(t_{i,k})} \rightarrow \overline{p_i : \mathbf{proc}(b_{i,k})} \in \Delta \quad \Delta; \overline{p_i.id_{i,j} : t_{i,j}} \vdash C : \{p_i : \mathbf{proc}(b_{i,k})\}}}{\Delta \vdash ID(\overline{p_i : \mathbf{proc}(id_{i,j} : t_{i,j})}) : \overline{p_i : \mathbf{proc}(b_{i,k})} \{C\}} \text{TDEF}$

■ **Figure 3** Modular Choreographies, typing.

Typing judgments are of four forms. A judgment $\gamma \vdash e : t$ indicates that the local expression e has type t under the local typing environment γ . A judgment $\Gamma \vdash \Sigma$ signifies that the memory state Σ is consistent with the declarations in Γ i.e., that maps each symbol defined in Γ to an element of the expected type. A judgment $\Delta; \Gamma \vdash R : S$ indicates that under Δ and Γ , the (runtime) choreography R has return type S . Rules TNIL and TRET type the exit points of a choreography and Rule TCOND combines the information about exit points in each branch (C_1, C_2) and in the continuation (C_3) ensuring that if values are returned they are all of the same type (by definition of \vee) and that the continuation is empty whenever all exit points in both branches return (i.e., $\perp \notin C_1 \vee C_2$). Rule TCALL ensures that the called function is declared and that the actual parameters ($p_i.e_{i,k}$) and target variables ($p_i.id_{i,j}$) are of the types specified by the function signature ($q_i.t_{i,k}$ for formal parameters and $q_i.b_{i,k}$ for results, respectively). Rule TCALL ensures that the type of the target variables ($p_i.id_{i,j}$) for the call under execution (R) coincides with the type of the values returned by (every) exit point for the call. The remaining inference rules for

$\Delta; \Gamma \vdash R: S$ follow the intuition behind the different choreographic primitives. Observe that rule selection is completely syntax-driven: the outermost construct of a term identifies uniquely identifies which rule to apply. Finally, a judgment $\Delta \vdash \mathcal{C}$ means that the function definitions in \mathcal{C} agree with the signatures declared in Δ and are well-typed. The last property is captured by Rule TDEF which states that the body of a choreographic function has the expected return type under Δ and the environment Γ given by its formal arguments.

In the remainder we assume subject reduction for local expressions i.e., that evaluating a well-typed expression in a memory state compatible with the environment used to type the expression yields a result of the expected type.

► **Assumption 3.** *If $\Gamma \vdash \Sigma$, $\Gamma(p) \vdash e: t$, and, $\Sigma(p) \vdash e \downarrow c$, then $\Gamma(p) \vdash c: t$.*

Under Assumption 3, well-typed configurations can only evolve into well-typed configurations i.e., MC enjoys subject reduction.

► **Theorem 4 (Subject reduction).** *If $\Delta \vdash \mathcal{C}$, $\Delta; \Gamma \vdash R: S$, $\Gamma \vdash \Sigma$ and $\langle \Gamma, R, \Sigma \rangle \rightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle$, then $\Gamma' \vdash \Sigma'$ and $\Delta; \Gamma' \vdash R': S'$ for $S' \subseteq S$.*

Proof. By nested induction on the derivation of $\Delta; \Gamma \vdash R: S$ and $\langle \Gamma, R, \Sigma \rangle \rightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle$.

- Consider the case of Rule TDEC. Then, $R = \mathbf{var} \ p.id: b; C$, $p.id \notin \Gamma$, and $\Delta; \Gamma \vdash C: S$. The only case for $\langle \Gamma, R, \Sigma \rangle \rightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle$ is that of Rule SDEC and thus $\Gamma' = \Gamma, p.id: b$, $R' = C$, and $\Sigma' = \Sigma[p.id := \text{default}(b)]$. By definition $\text{default}(b) \in \text{Values}(b)$ and thus $\Gamma' \vdash \Sigma'$.
- Consider the case of Rule TCOM. Then, $R = p.e \rightarrow q.id; C$, $\Gamma(p) \vdash e: b$, $\Gamma(q) \vdash id: b$, and $\Delta; \Gamma \vdash C: S$. The only case for $\langle \Gamma, R, \Sigma \rangle \rightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle$ is that of Rule SCOM and thus $\Gamma' = \Gamma$, $R' = C$, and $\Sigma' = \Sigma[q.id := v]$ where $\Sigma(p) \vdash e \downarrow v$. It follows from $\Gamma(p) \vdash e: b$ and Assumption 3 that $\Gamma' \vdash \Sigma'$.
- Consider the case of Rule TCOND. Then, $R = \mathbf{if} \ p.e \{C_1\} \ \mathbf{else} \ \{C_2\}; C_3$, $\Gamma(p) \vdash e: \text{boolean}$, and $\Delta; \Gamma \vdash C_i: S_i$ for $i \in \{1, 2, 3\}$. It follows from $\Gamma(p) \vdash e: \text{boolean}$, $\Gamma \vdash \Sigma$ and Assumption 3 that $\Sigma(p) \vdash e \downarrow v$ for $v \in \text{Values}(\text{boolean})$ and thus that the only case for $\langle \Gamma, R, \Sigma \rangle \rightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle$ is that of Rule SCOND. If $\Sigma(p) \vdash e \downarrow \text{true}$ then $R' = C_1 \wp C_3$, $S' = S_1 \vee S_3$, $\Gamma' = \Gamma$, $\Sigma' = \Sigma$. To derive $\Delta; \Gamma' \vdash R': S'$ it suffices to take all applications Rule TNIL in the derivation of $\Delta; \Gamma \vdash C_1: S_1$ that correspond to an occurrence of ε replaced with C_3 in $C_1 \wp C_3$ and replace them with a copy of the derivation for $\Delta; \Gamma \vdash C_3: S_3$. By definition of \vee , $S' \subseteq S$. The case for $\Sigma(p) \vdash e \downarrow \text{false}$ is similar.
- Consider the case of Rule TCALLRT. Then, $R = \overline{p_i.(id_{i,j})} = \langle \Gamma_1, R_1, \Sigma_1 \rangle; C$, $\Delta; \Gamma_1 \vdash R_1: \{p_i: \mathbf{proc}(\overline{b_{i,j}})\}$, $\Gamma_1 \vdash \Sigma_1$, $\Gamma \vdash p_i.id_{i,j}: \overline{b_{i,j}}$, and $\Delta; \Gamma \vdash C: S$. Any derivation of $\langle \Gamma, R, \Sigma \rangle \rightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle$ starts with an application of either Rule SCALLRET or Rule SCALLRT.
 - In the first case, $R_1 = \mathbf{return} \ \overline{p_i.(e_{i,j})}$, $R' = C$, $\Gamma' = \Gamma$, $\Sigma' = \Sigma[\overline{p_i.id_{i,j}} := \overline{v_{i,j}}]$ where $\overline{\Sigma_1(p_i)} \vdash e_{i,j} \downarrow v_{i,j}$, and $S' = S$. By $\Delta; \Gamma_1 \vdash R_1: \{p_i: \mathbf{proc}(\overline{b_{i,j}})\}$ and Assumption 3, $v_{i,j} \in \text{Values}(b_{i,j})$ and thus $\Gamma' \vdash \Sigma'$.
 - In the second case, $\langle \Gamma_1, R_1, \Sigma_1 \rangle \rightarrow_{\mathcal{C}} \langle \Gamma_2, R_2, \Sigma_2 \rangle$, $R' = \overline{p_i.(id_{i,j})} = \langle \Gamma_2, R_2, \Sigma_2 \rangle; C$, $\Gamma' = \Gamma$, $\Sigma' = \Sigma$, and $S' = S$. By inductive hypothesis, $\Gamma_2 \vdash \Sigma_2$ and $\Delta; \Gamma_2 \vdash R_2: \{p_i: \mathbf{proc}(\overline{b_{i,j}})\}$ and by Rule TCALLRT, $\Delta; \Gamma' \vdash R': S'$.
- Remaining cases are straightforward adaptations of the ones above. ◀

Well-typed choreographies enjoy progress: a choreography equipped with memory compatible with its typing environment can always perform a transition unless it is terminated (it consists of a return instruction or no instruction at all).

► **Theorem 5** (Progress). *If $\Delta \vdash C$, $\Delta; \Gamma \vdash R: S$, $\Gamma \vdash \Sigma$, then either*

- $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma', R', \Sigma' \rangle$ or
- R is either ε or a return (i.e., has form **return** $\overline{p.(\bar{e})};$).

Proof. By induction on the derivation for $\Delta; \Gamma \vdash R: S$.

- Consider the case of Rule TNIL. Then, $R = \varepsilon$ and $\langle \Gamma, \varepsilon, \Sigma \rangle$ does not admit any transition.
- Consider the case of Rule TRET. Then, $R = \mathbf{return} \overline{p.(\bar{e})};$ and $\langle \Gamma, R, \Sigma \rangle$ does not admit any transition.
- Consider the case of Rule TDEC. Then, $R = \mathbf{var} \overline{p.id:b}; C$, $p.id \notin \Gamma$, and $\Delta; \Gamma \vdash C: S$. By Rule SDEC, $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma', C, \Sigma' \rangle$ where $\Gamma' = \Gamma, p.id: b$ and $\Sigma' = \Sigma[p.id := \text{default}(b)]$.
- Consider the case of Rule TCOM. Then, $R = p.e \rightarrow q.id; C$, $\Gamma(p) \vdash e: b$, $\Gamma(q) \vdash id: b$, and $\Delta; \Gamma \vdash C: S$. It follows from $\Gamma(p) \vdash e: b$, $\Gamma \vdash \Sigma$ and Assumption 3 that $\Sigma(p) \vdash e \downarrow v$ for $v \in \text{Values}(b)$. By Rule SCOM, $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma, C, \Sigma' \rangle$ where $\Sigma' = \Sigma[q.id := v]$.
- Consider the case of Rule TCOND. Then, $R = \mathbf{if} \overline{p.e} \{C_1\} \mathbf{else} \{C_2\}; C_3$, $\Gamma(p) \vdash e: \mathbf{boolean}$, and $\Delta; \Gamma \vdash C_i: S_i$ for $i \in \{1, 2, 3\}$. It follows from $\Gamma(p) \vdash e: \mathbf{boolean}$ and $\Gamma \vdash \Sigma$ that $\Sigma(p) \vdash e \downarrow v$ for $v \in \text{Values}(\mathbf{boolean})$. By Rule SCOND, $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma, R', \Sigma' \rangle$ and where R' is $C_1 \mathbin{\&} C_3$ if $\Sigma(p) \vdash e \downarrow \mathbf{true}$ and $C_2 \mathbin{\&} C_3$ otherwise.
- Consider the case of Rule TCALL. Then, $R = \overline{p_i.(id_{i,j})} = ID(\overline{p_i.(e_{i,k})}); C$, $\Delta; \Gamma \vdash C: S$, $\Delta = \Delta'$, $ID: \overline{q_i:\mathbf{proc}(t_{i,k})} \rightarrow \overline{q_i:\mathbf{proc}(b_{i,j})}$, $\overline{\Gamma(p_i) \vdash e_{i,k}: t_{i,k}}$, and $\overline{\Gamma(p_i) \vdash id_{i,j}: b_{i,j}}$. It follows that $ID(\overline{q_i:\mathbf{proc}(id_{i,k}:t_{i,k})}): (\overline{q_i:\mathbf{proc}(id_{i,j}:b_{i,j})})\{C'\} \in C$, $\overline{\Sigma(p_i) \vdash e_{i,k} \downarrow c_{i,k}}$, and $\overline{\Gamma(p_i) \vdash c_{i,k}: t_{i,k}}$. By Rule SCALL, there is $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma, R', \Sigma \rangle$ with $R' = \overline{p_i.(id_{i,j})} = \langle \overline{p_i.id_{i,k}: t_{i,k}}, C'[p_i/q_i], \Sigma[p_i.id_{i,k} := c_{i,k}] \rangle; C$.
- Consider the case of Rule TCALLRT. Then, $R = \overline{p_i.(id_{i,j})} = \langle \Gamma_1, R_1, \Sigma_1 \rangle; C$, $\Delta; \Gamma_1 \vdash R_1: \{ \overline{p_i:\mathbf{proc}(b_{i,j})} \}$, $\Gamma_1 \vdash \Sigma_1$, $\overline{\Gamma \vdash p_i.id_{i,j}: b_{i,j}}$, and $\Delta; \Gamma \vdash C: S$. Because of its type, R_1 cannot be ε . By inductive hypothesis, either $\langle \Gamma_1, R_1, \Sigma_1 \rangle \longrightarrow_C \langle \Gamma_2, R_2, \Sigma_2 \rangle$ or $R_1 = \mathbf{return} \overline{p_i.(\bar{e}_{i,j})};$. Assume the first case. By Rule SCALLRT, $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma, R', \Sigma \rangle$ for $R' = \overline{p_i.(id_{i,j})} = \langle \Gamma_2, R_2, \Sigma_2 \rangle; C$. Assume the second case. By hypothesis, $\Sigma_1(p_i) \vdash e_{i,j} \downarrow v_{i,j}$ and by Rule SCALLRET, $\langle \Gamma, R, \Sigma \rangle \longrightarrow_C \langle \Gamma, C, \Sigma[p_i.id_{i,j} := v_{i,j}] \rangle$.
- Remaining cases are straightforward adaptations of the ones above. ◀

4 Translation to Choral

In this section we describe how our tool can translate choreographies in our language to Java implementations via the Choral language [20].

Choral is an object-oriented choreographic programming language: in Choral classes and interfaces represent distributed data types parametric in the processes participating in them. For instance, the snippet below contains the definition of a Choral class `Tuple2` that implements a generic tuple distributed over two processes (`A` and `B`) each holding one of the two components of the tuple (`left` is stored at `A` and `right` at `B`).

```
public class Tuple2@A(B)<L@C,R@D> {
  public final L@A left; public final R@B right;
  public Tuple2(L@A left, R@B right) {
    this.left = left; this.right = right;
  }
}
```

Choral Code

The Choral compiler generate Java implementation for each participant of a Choral type. For instance, `Tuple2` is compiled to the following Java classes.

3:12 Modular Choreographies

```
// Implementation of Tuple2 for A
public class Tuple2_A<L,R> {
    public final L left;
    public Tuple2_A(L left) {
        this.left = left;
    }
}
```

Java Code

```
// Implementation of Tuple2 for B
public class Tuple2_B<L,R> {
    public final R right;
    public Tuple2_B(R right) {
        this.right = right;
    }
}
```

Java Code

Types like `Tuple2` above allow us to represent in Choral the distributed return of MC. For instance, `return (a.true,b.5)` is translated into `return new Tuple2@A,B<Boolean,Integer>(true@A,5@B)`.

Differently from other choreographic languages like MC, Choral does not fix a communication primitive. Instead, communication can be programmed directly: The Choral standard library provides a framework with several kinds of channels but programmers can provide their own by writing them directly in Choral or by wrapping implementations written in Java into Choral types. For instance, the Choral standard library defines the following interface to represent a generic channel between two processes, abstracted by `A` and `B`, for transmitting data of a given type, abstracted by the type parameter `T`.

```
public interface SymDataChannel@A,B<T@C> {
    public <M@C extends T@C> M@B com(M@A message);
    public <M@C extends T@C> M@A com(M@B message);
}
```

Choral Code

Data transmission is performed by invoking the (overloaded) method `com` which takes any value of a subtype `M` of `T` located at one process, say `A`, and returns a value of the same type at the other process, say `B`. In the example below, `A` transmits `5` to `B` using the channel `chAB`.

```
SymDataChannel@A,B<Serializable> chAB = /* ... */
Integer@B x = chAB.<Integer>com(5@A);
```

Choral Code

The interface `SymChannel` extends `SymDataChannel` with methods for selecting labels (which are represented as enums).

```
public interface SymChannel@A,B<T@C> extends SymDataChannel@A,B<T> {
    public <L@C extends Enum@C<L>> L@B select(L@A label);
    public <L@C extends Enum@C<L>> L@A select(L@B label);
}
```

Choral Code

Building on these features of Choral (and standard constructs like assignments and conditionals) we can translate any choreography function written in MC into Choral. Given a choreography function, our translation generates a Choral class with the same name and parametrised in the participants of the function. For instance, the choreographic function `DH` from Example 1 is compiled to a Choral class `DH@A,B`, as shown in the example below. This class exposes a single static method `run` which implements the behaviour of the choreographic function. In addition to the parameters specified by the choreography function, `run` takes a `SymChannel` for each pair of participants (we follow a lexicographic order to avoid ambiguity). The body of the method is generated by mapping communications and selections to invocations of the methods exposed by these channels, and other constructs homomorphically. We illustrate the translation in the examples below, we include the original MC line as comments for readability.

► **Example 6.** Consider the choreographic function DH from Example 1, our tool generates the following Choral implementation for it (comments are added for readability).

```
public class DH@A,B {
  public static Tuple2@A,B<Integer,Integer> run(
    /* channel between A and B */
    SymChannel@A,B<Serializable> chAB,
    /* parameters for A: proc(privKey:int,g:int,m:int,exp:(int,int,int)->int) */
    Integer@A privKey_A, Integer@A g_A, Integer@A m_A,
    Function3@A<Integer,Integer,Integer,Integer> exp_A,
    /* parameters for B: proc(privKey:int,g:int,m:int,exp:(int,int,int)->int) */
    Integer@B privKey_B, Integer@B g_B, Integer@B m_B,
    Function3@B<Integer,Integer,Integer,Integer> exp_B
  ) {
    // a computes and sends its public key to b
    // a.exp(privKey, g, m) -> var b.pubKey:int;
    Integer@B pubKey_B = chAB.<Integer>com( exp_A(privKey_A, g_A, m_A) );
    // b computes and sends its public key to a
    // b.exp(privKey, g, m) -> var a.pubKey:int;
    Integer@A pubKey_A = chAB.<Integer>com( exp_B(privKey_B, g_B, m_B) );
    // a computes its copy of the shared secret
    // var a.key:int = exp(privKey, pubKey, m);
    Integer@A key_A = exp_A(privKey_A, pubKey_A, m_A );
    // b computes its copy of the shared secret
    // var b.key:int = exp(privKey, pubKey, m);
    Integer@B key_B = exp_B(privKey_B, pubKey_B, m_B );
    // a and b return the shared secret
    // return a.key, b.key;
    return new Tuple2@A,B<Integer,Integer>(key_A, key_B);
  }
}
```

Choral Code

► **Example 7.** Consider the choreographic function SS0 from Example 2, our tool generates the following implementation in Choral for it.

```
public class SS0@C, S, A {
  public static Integer@C run(
    /* channels */
    SymChannel@A, C<Serializable> chAC,
    SymChannel@C, S<Serializable> chCS,
    /* parameters for C: proc(creds:()->int) */
    Supplier@C<Integer> creds_C,
    /* parameters for S: proc(newToken:()->int) */
    Supplier@S<Integer> newToken_S,
    /* parameters for A: proc(valid:(int)->boolean) */
    Function@A<Integer, Boolean> valid_A,
  ) {
    // c.creds() -> var a.x:int
    Integer@A x = chAC.<Integer>com( creds_C() );
    // var c.t:int;
    Integer@C t_C;
    // if a.valid(x)
    if (valid_A(x)) {
      // a -> s[OK];
      chAS.<Label>select(Label@A.OK);
      // s -> c[TOKEN];
      chCS.<Label>select(Label@S.TOKEN);
      // s.newToken() -> c.t;
      t_C = chCS.<Integer>com( newToken_S() );
    } else {
      // a -> s[KO];
      chAS.<Label>select(Label@A.KO);
      // s -> c[ERROR];
    }
  }
}
```

Choral Code

3:14 Modular Choreographies

```
chCS.<Label>select(Label@S.ERROR);
// c.t = SSO(c.creds, s.newToken, a.valid);
t_C = SSO@(C, S, A).run(chAC, chCS, creds_C, newToken_S, valid_A);
}
// return c.t;
return t_C;
}
}
```

Choral Code

The Choral code generated by our translation is correct and well-typed, provided the original choreography is also well-typed. To enforce this requirement, our tool implements the typing discipline described in Section 3. First, the typing environment Δ is populated using the signatures of the function definitions provided to the tool. Then, each procedure definition is checked as specified by Rule TDEF using the syntax to guide the selection of the corresponding typing rule.

To obtain Java implementations, we then rely on the Choral compiler.

► **Example 8.** Continuing Example 6, the Choral compiler produces the following implementation for *A* (the one for *B* is similar).

```
public class DH_A {
    public static Tuple2_A<Integer,Integer> run(
        SymChannel_A<Serializable> chAB,
        Integer privKey_A, Integer g_A, Integer m_A,
        Function3<Integer,Integer,Integer,Integer> exp_A
    ) {
        chAB.com<Integer>( exp_A(privKey_A,g_A,m_A) );
        Integer pubKey_A = chAB.com<Integer>();
        Integer key_A = exp_A(privKey_A,pubKey_A,m_A) );
        return new Tuple2_A<Integer,Integer>(key_A);
    }
}
```

Java Code

► **Example 9.** Continuing Example 7, the Choral compiler produces the following implementations for *A*, *C*, and *S*.

```
public class SSO_A {
    public static void run(
        SymChannel_A<Serializable> chAC,
        SymChannel_A<Serializable> chAS,
        Function<Integer,Boolean> valid_A
    ) {
        Integer x = chAC.<Integer>com();
        if ( valid_A(x) ) {
            chAS.<Label>select(Label.OK);
        } else {
            chAS.<Label>select(Label.KO);
            SSO.run(chAC, chAS, valid_A);
        }
        return;
    }
}
```

Java Code

```

public class SSO_C {
    public static Integer run(
        SymChannel_B<Serializable> chAC,
        SymChannel_A<Serializable> chCS,
        Supplier<Integer> creds_C
    ) {
        chAC.<Integer>com( creds_C() );
        Integer t_C;
        switch ( chCS.<Label>select() ):
            case Label.TOKEN -> {
                t_C = chCS.<Integer>com();
            }
            case Label.ERROR -> {
                t_C = SSO.run(chAC, chCS,
                    creds_C);
            }
        }
        return t_C;
    }
}

```

Java Code

```

public class SSO_S {
    public static void run(
        SymChannel_B<Serializable> chAS,
        SymChannel_B<Serializable> chCS,
        Supplier<Integer> newToken_S
    ) {
        switch ( chAS.<Label>select() ):
            case Label.OK -> {
                chCS.<Label>select(Label.TOKEN);
                chCS.<Integer>com( newToken_S()
                    );
            }
            case Label.KO -> {
                chCS.<Label>select(Label.ERROR);
                SSO.run(chAS, chCS, newToken_S);
            }
        }
        return;
    }
}

```

Java Code

5 Conclusion and Future Work

Modular Choreographies brings the simplicity of choreographic programming based on the “Alice and Bob” communication abstraction one step nearer to practical programming.

An immediate direction for future work will be extending our language with more features, trying to keep the syntax as simple as possible. Inspiration for this could naturally come from theoretical models of choreographic languages, as those found in the research lines of choreographic programming and multiparty session types (abstract choreographies without computation) [32, 23, 24, 1]. For example, adding features for dynamic topologies (e.g., spawning new processes) is important for capturing some parallel algorithms [10, 34, 42], nondeterminism is crucial to capturing patterns like barriers and producers/consumers [32], and mixed choices or unordered communication sets are relevant for modelling exchanges [32, 36, 8, 13]. Another source of inspiration might come from the developments of choreographic programming languages like Choral, whose expressivity benefits significantly from the integration with mainstream programming abstractions like functions and objects (and their related type theories). For example, Choral is expressive enough to implement full-duplex asynchronous communications, where interactions can be triggered by any participant and be interleaved freely [30]. Another interesting line of future work regards formalising the guarantees provided by Modular Choreographies. Choreographic programming languages equipped with general recursion, like ours, are known to provide deadlock-freedom [32]. To prove this with confidence, one could extend previous formalisations of choreographic programming in theorem provers, as those given in [15, 14, 16, 38, 22].

References

- 1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/25000000031.

- 2 Alessandro Bruni, Marco Carbone, Rosario Giustolisi, Sebastian Mödersheim, and Carsten Schürmann. Security protocols as choreographies. In Daniel Dougherty, José Meseguer, Sebastian Alexander Mödersheim, and Paul D. Rowe, editors, *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday*, volume 13066 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2021. doi:10.1007/978-3-030-91631-2_5.
- 3 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012. doi:10.1145/2220365.2220367.
- 4 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.
- 5 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:24)2012.
- 6 Edmund M. Clarke, William Klieber, Milos Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.
- 7 Luís Cruz-Filipe, Eva Graversen, Fabrizio Montesi, and Marco Peressotti. Reasoning about choreographic programs. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13908 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2023. doi:10.1007/978-3-031-35361-1_8.
- 8 Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. In Javier Esparza and Andrzej S. Murawski, editors, *FoSSaCS*, volume 10203 of *LNCS*, pages 424–440. Springer, 2017. doi:10.1007/978-3-662-54458-7_25.
- 9 Luís Cruz-Filipe, Lovro Lugovic, and Fabrizio Montesi. Certified compilation of choreographies with hacc. In Marieke Huisman and António Ravara, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13910 of *Lecture Notes in Computer Science*, pages 29–36. Springer, 2023. doi:10.1007/978-3-031-35355-0_3.
- 10 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In *FORTE*, volume 9688 of *LNCS*, pages 114–123. Springer, 2016. doi:10.1007/978-3-319-39570-8_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 and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/J.TCS.2019.07.005.
- 13 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.
- 14 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan*,

- September 8-10, 2021, *Proceedings*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.
- 15 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a turing-complete choreographic language in coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPICs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ITP.2021.15.
 - 16 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. doi:10.1007/S10817-023-09665-3.
 - 17 Mila Dalla Preda, Maurizio Gabbriellini, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.
 - 18 Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.
 - 19 Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017. doi:10.1007/978-3-319-67425-4_12.
 - 20 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst.*, Nov 2023. Accepted. doi:10.1145/3632398.
 - 21 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty languages: The choreographic and multitier cases (pearl). In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPICs*, pages 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ECOOP.2021.22.
 - 22 Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi:10.1145/3498684.
 - 23 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016. doi:10.1145/2827695.
 - 24 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
 - 25 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies—computing preconditions in choreographic programming. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Proceedings*, volume To appear of *Lecture Notes in Computer Science*. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.
 - 26 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016. doi:10.1145/2872362.2872374.
 - 27 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.
 - 28 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

- 29 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016. doi:10.1007/978-3-319-39570-8_13.
- 30 Lovro Lugovic and Fabrizio Montesi. Real-world choreographic programming: An experience report. *CoRR*, abs/2303.03983, 2023. doi:10.48550/ARXIV.2303.03983.
- 31 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. URL: <https://www.fabriziomontesi.com/files/choreographic-programming.pdf>.
- 32 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.
- 33 R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, dec 1978. doi:10.1145/359657.359659.
- 34 Nicholas Ng and Nobuko Yoshida. Pabble: parameterised scribble. *Serv. Oriented Comput. Appl.*, 9(3-4):269–284, 2015. doi:10.1007/S11761-014-0172-8.
- 35 Peter W. O’Hearn. Experience developing and deploying concurrency analysis at facebook. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2018. doi:10.1007/978-3-319-99725-4_5.
- 36 Kirstin Peters and Nobuko Yoshida. On the expressiveness of mixed choice sessions. In Valentina Castiglioni and Claudio Antares Mezzina, editors, *Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics, EXPRESS/SOS 2022, and 19th Workshop on Structural Operational Semantics Warsaw, Poland, 12th September 2022*, volume 368 of *EPTCS*, pages 113–130, 2022. doi:10.4204/EPTCS.368.7.
- 37 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, MA, USA, 2002.
- 38 Johannes Åman 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.
- 39 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPICs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ECOOP.2017.24.
- 40 Gan Shen, Shun Kashiwa, and Lindsey Kuper. Haschor: Functional choreographic programming for all (functional pearl). *CoRR*, abs/2303.00924, 2023. doi:10.48550/ARXIV.2303.00924.
- 41 Ton Smeele and Sung-Shik Jongmans. Choreographic programming of isolated transactions. *Electronic Proceedings in Theoretical Computer Science*, 378:49–60, apr 2023. doi:10.4204/EPTCS.378.5.
- 42 Vasco T. Vasconcelos, Francisco Martins, Hugo-Andrés López, and Nobuko Yoshida. A type discipline for message passing parallel programs. *ACM Trans. Program. Lang. Syst.*, 44(4):26:1–26:55, 2022. doi:10.1145/3552519.
- 43 Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Comput. Surv.*, 53(4):81:1–81:35, 2020. doi:10.1145/3397495.