Choral: Object-oriented Choreographic Programming

SAVERIO GIALLORENZO, Università di Bologna, Italy and INRIA, France
FABRIZIO MONTESI and MARCO PERESSOTTI, University of Southern Denmark, Denmark

Choreographies are coordination plans for concurrent and distributed systems, which define the roles of the involved participants and how they are supposed to work together. In the paradigm of choreographic programming, choreographies are programs that can be compiled into executable implementations.

In this article, we present Choral, the first choreographic programming language based on mainstream abstractions. The key idea in Choral is a new notion of data type, which allows for expressing that data is distributed over different roles. We use this idea to reconstruct the paradigm of choreographic programming through object-oriented abstractions. Choreographies are classes, and instances of choreographies are objects with states and behaviours implemented collaboratively by roles.

Choral comes with a compiler that, given a choreography, generates an implementation for each of its roles. These implementations are libraries in pure Java, whose types are under the control of the Choral programmer. Developers can then modularly compose these libraries in their programs, to participate correctly in choreographies. Choral is the first incarnation of choreographic programming offering such modularity, which finally connects more than a decade of research on the paradigm to practical software development.

The integration of choreographic and object-oriented programming yields other powerful advantages, where the features of one paradigm benefit the other in ways that go beyond the sum of the parts. On the one hand, the high-level abstractions and static checks from the world of choreographies can be used to write concurrent and distributed object-oriented software more concisely and correctly. On the other hand, we obtain a much more expressive choreographic language from object-oriented abstractions than in previous work. This expressivity allows for writing more reusable and flexible choreographies. For example, object passing makes Choral the first higher-order choreographic programming language, whereby choreographies can be parameterised over other choreographies without any need for central coordination. We also extend method overloading to a new dimension: specialisation based on data location. Together with subtyping and generics, this allows Choral to elegantly support user-defined communication mechanisms and middleware.

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; • **Software and its engineering** → **Multiparadigm languages**; **Classes and objects**; *Concurrent programming structures;*

Additional Key Words and Phrases: Choreographies, communication, higher-kinded types

ACM Reference format:
Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (January 2024), 59 pages.
https://doi.org/10.1145/3632398

This work was partially supported by Villum Fonden, Grant No. 29518.
Work mainly performed while S. Giallorenzo was employed at the University of Southern Denmark.
Authors' addresses: S. Giallorenzo, Department of Computer Science and Engineering, Universitá di Bologna, Mura Anteo Zamboni 7, Bologna, 40126, Italy, INRIA, Sophia-Antipolis, France; e-mail: saverio.giallorenzo@gmail.com; F. Montesi (Corresponding author) and M. Peressotti, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, Odense, 5230, Denmark; e-mails: {fmontesi, peressotti}@imada.sdu.dk.

0164-0925/2024/01-ART1 $15.00
https://doi.org/10.1145/3632398

1 INTRODUCTION

Background. Choreographies, broadly construed, are coordination plans for concurrent and distributed systems [Object Management Group 2011; W3C 2004]. Examples of choreographies include distributed authentication protocols [OpenID Foundation 2014; Sporny et al. 2011], cryptographic protocols [Diffie and Hellman 1976], and multiparty business processes [Object Management Group 2011; W3C 2004]. In software development, programmers use choreographies to agree on the interactions that communicating endpoints should enact to achieve a common goal; then, each endpoint can be programmed independently. The success of this development process hinges on achieving *choreography compliance*: when all endpoints are run together, they interact as defined by the choreographies agreed upon [Montesi 2023].

Achieving choreography compliance is hard, because of some usual suspects of concurrent and distributed programming: predicting how multiple programs will interact at runtime is challenging [O'Hearn 2018], and mainstream programming languages do not adequately support programmers in reasoning about coordination in their code [Leesatapornwongsa et al. 2016; Lu et al. 2008]. Additionally, choreographies are complex. At a minimum, choreographies define the expected communication flows among their roles (abstractions of endpoints, like "Alice," "Bob," "Buyer," etc.) [Intl. Telecommunication Union 1996]. However, they often include computational details of arbitrary complexity, for example, pre- or post-processing of data (encryption, validation, anonymisation, etc.), state information, and decision procedures to choose among alternative behaviours. These computational details are essential parts of many protocols, ranging from security protocols to business processes. Figure 1 depicts the common situation where a programmer tries to ensure choreography compliance through their subjective interpretation of the choreography and the manual coding of endpoints.

In response to the challenge of choreography compliance, researchers investigated methods to relate choreographies to endpoint programs—many are reviewed in Ancona et al. [2016]; Hüttel et al. [2016]. Initially, these methods focused on simple choreographic languages without computation that were inspired by, e.g., communicating automata, process calculi, and session types [Basu et al. 2012; Bravetti and Zavattaro 2007; Honda et al. 2016; Qiu et al. 2007]. Some ideas developed for choreographies were later adopted in the paradigm of *choreographic programming* [Carbone and Montesi 2013; Montesi 2013], where choreographies are written in a Turing-complete programming language that allows for defining arbitrary computation at endpoints. Thanks to the capability of combining computation with coordination, choreographic programming languages can capture realistic protocols that include data manipulation and decision procedures, including encryption strategies (as in security protocols), retry strategies (as in transport protocols), and marshalling (as in application protocols). In choreographic programming, compliance is obtained by construction: given a choreography, a compiler automatically translates it to a set of compliant endpoint implementations. Choreographic programming has been shown to have promising potential in multiple contexts, including information flow [Lluch-Lafuente et al. 2015], distributed algorithms [Cruz-Filipe and Montesi 2016], cyber-physical systems [López and Heussen 2017; López et al. 2016], and self-adaptive systems [Dalla Preda et al. 2017].

The problem. Current approaches to choreographic programming come at a significant cost to modularity, and it remains unclear how the benefits of this paradigm can be applied to mainstream software development.

Typically, the endpoint code that implements a choreography comes in libraries that developers can use in their applications through an Application Programming Interface (API) [Atzori et al. 2010; Murty 2008; Wilder 2012]. For example, a library that implements a choreography for user authentication might provide a method `authenticate` that a web service can invoke

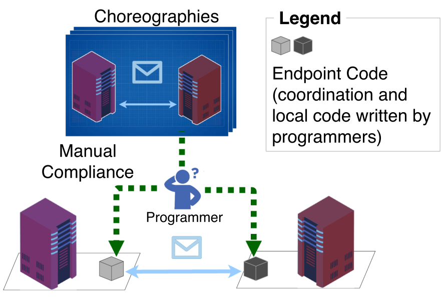

Choreographies
Legend
Endpoint Code (coordination and local code written by programmers)
Manual Compliance
Programmer

Fig. 1. Choreography compliance: Endpoints should communicate as intended by the choreographies that they engage in.

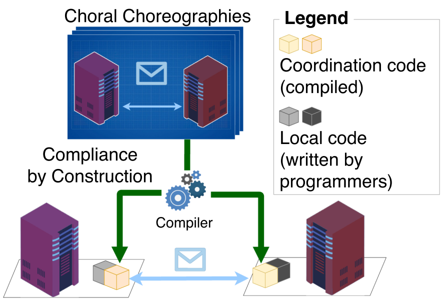

Choral Choreographies
Legend
Coordination code (compiled)
Local code (written by programmers)
Compliance by Construction
Compiler

Fig. 2. Our proposal: Compliant-by-construction libraries are automatically compiled from choreographies in Choral.

to run (its part of) the protocol in collaboration with a connected client. The implementation of `authenticate` might involve a series of communications between the client and the web service. Potentially, a third-party identity provider might be involved as well, and messages might include passwords that should be hashed. Usually, and ideally, all these details are hidden from the API (the signature of the method) exposed to the developer of the web service. This abstraction allows the developer to minimise coupling between their implementation of the functionalities offered by the web service and the choreography used for authentication, bringing the expected benefit: the library implementing the choreography and the rest of the code of the web service can be updated independently and recombined, as long as the API provided by the former remains compatible with the one expected by the latter. This benefit is key to large-scale software development, and it is an initial assumption for modern development practices like microservices and DevOps [Dragoni et al. 2017].

Unfortunately, previous frameworks for choreographic programming do not support modular software development [Dalla Preda et al. 2017; Montesi 2013]. The code that these frameworks generate for each endpoint is a "black box" program without an API: it can be executed, but it is not designed to be composed by programmers within larger codebases. Thus, these frameworks fall short of providing the aforementioned benefit of modularity. Furthermore, the common scenario of programming an endpoint that participates in multiple choreographies is not supported, and neither is programming an endpoint where participating in a choreography is only part of what the endpoint does.

A major factor that makes modularity challenging is that current choreographic languages are based on behavioural models (process calculi, communicating automata, etc.), which makes translating choreographies into libraries based on mainstream abstractions (data, functions, objects, etc.) nontrivial. For the simpler setting of choreographic languages without computation, we know that choreographies can be translated to libraries that offer a "fluid" API. For instance, Scalas et al. [2017] produce object-oriented libraries whose APIs enforce the invocation of `send` and `receive` methods in the right order: If an endpoint should send, receive, and then send, then the developer will be forced to write something like `o.send(..).receive(..).send(..)`. However, this approach leaks the structure of the communication flow implemented by the library; thus, future versions of the library that adopt a different structure (e.g., an unnecessary communication might be removed, or some communications might be bunched together for efficiency) would require rewriting the application that uses the library. Furthermore, this approach does not let the choreography programmer decide how the generated API will look; thus, we cannot use this method to generate drop-in replacements for existing libraries.

In summary, we still have to discover how the principles of choreographic programming can be applied to mainstream programming. This article aims to fill this gap.

This article. We present Choral, a new choreographic programming language that supports modularity and is based on mainstream programming concepts (from object-oriented programming). To demonstrate applicability, Choral is compatible with Java, but our ideas apply to most statically typed, object-oriented languages.

The fulcrum of Choral is a new interpretation of choreographies that builds naturally on top of existing language abstractions: Choral is an *object-oriented* choreographic programming language, where choreographies are classes and their instances are objects. The starting point for this interpretation is a generalisation of the key idea found in Lambda 5 [Murphy VII et al. 2004], the model that inspired the research line on multitier programming [Cooper et al. 2006; Murphy VII et al. 2007; Neubauer and Thiemann 2005; Serrano et al. 2006; Weisenburger et al. 2020]. In Lambda 5, each data type is located at a (single) place, which enables reasoning on spatially distributed computation. Choral generalises these types from single to *multiple* locations, which allows us to express that an object is implemented choreographically: Choral objects have types of the form `T@(A1, ..., An)`, where `T` is the usual interface of the object, and `A1`, ..., `An` are the roles that collaboratively implement the object.

As an example, consider the case of a multiparty choreography for distributed authentication, where a service authenticates a client via a third-party identity provider. We can define such a choreography as a Choral class with type `DistAuth@(Client, Service, IP)` (`IP` is short for identity provider). The class can implement methods that involve multiple roles. For example, it can offer a method `authenticate` with the following signature.

```
Optional@Service<AuthToken> authenticate(Credentials@Client credentials)
```
Choral Code

Invoking `authenticate` with some `Credentials` located at `Client` returns an authorisation token at `Service` (`Optional` since authentication can fail), denoting the movement of data.

We leverage our object-oriented interpretation of choreographies to develop a methodology for choreography compliance that supports modularity and is compatible with mainstream programming. We depict this methodology in Figure 2. Given the code of a Choral class with some roles, a compiler produces a compliant-by-construction software library in pure Java for each role ("coordination code" in Figure 2): each library contains the local implementation of what its role should do to execute the choreography correctly. These libraries offer Java APIs derived from the source choreographies, which reveal only the details pertaining to the implemented role. When a software developer programs an endpoint that should engage in a choreography, they can just take the library compiled for the role that their endpoint needs to play and use the library through its Java API. Those APIs allow developers to modularly compose multiple libraries within their code ("local code" in Figure 2), thus gaining the ability to participate in multiple choreographies.

The Java code compiled from the method `authenticate` for the roles `Client` and `Service` has the following signatures, respectively.

```
// Compiled code for Client
Unit authenticate(Credentials credentials)
```
Generated Code

```
// Compiled code for Service
Optional<AuthToken> authenticate()
```
Generated Code

The compiled signatures follow the principle that we have just mentioned—that is, for each role, only what pertains to that role is reported. This is why the signature compiled for `Client` has a parameter of type `Credentials` and return type `Unit`: in the source choreography, the parameter is located at `Client` whereas the return type is located at another role.[1] Following the same principle, the signature for `Service` has return type `Optional<AuthToken>` and no parameters.

Contributions. We outline our main contributions.

Language. We present Choral, the first choreographic programming language based on mainstream abstractions and interoperable with a mainstream language (Java). The key novelty of the Choral language is that data types are higher-kinded types parameterised on roles. We leverage this feature to bring the key aspects of choreographies to object-oriented programming (spatial distribution, interaction, and knowledge of choice). Choral is also the first truly higher-order choreographic language, where choreographies passed as arguments can carry state and invocations of higher-order choreographies require no centralised coordination [Dalla Preda et al. 2017; Demangeon and Honda 2012].

Integrating object-oriented principles with choreographies brings key benefits in the other direction too, in the sense that we gain a much richer choreographic language than the state-of-the-art. By using subtyping, we can define abstract APIs for choreographies that can be implemented in different ways and communication behaviours, bringing the usual substitution principle for objects to choreographies. Method overloading allows us to specialise computation based on the locations of arguments—which is a new dimension for overloading. Semantic parametricity enables the writing of reusable choreographies that treat uniformly parameters that implement a shared API. These features allow us to generalise choreographies from assuming a fixed communication primitive to user-definable communication methods, thus freeing Choral from commitments to any communication technology or middleware. Furthermore, we can define in Choral the first hierarchy of "channel types" for choreographies, which can be used to represent at the level of types the topological assumptions of a choreography. In general, users are free to define other 'channel types' to support more topologies.

We implement a type checker that, in addition to the expected checks for an object-oriented language, detects coding errors related to roles. For example, our checker can rule out computation at a role that erroneously accesses data at another role without proper communication. This makes distribution errors manifest to the programmer. Our typing also supports the reuse of all existing Java classes and interfaces in Choral, because every such type can be lifted to a Choral type located at a single role.

In Choral, choreographies are concrete software artefacts. This poses questions such as what code should go in these artefacts, what code should remain local, and how these two parts of software should interact through APIs. We elicit these questions and report on our experience in addressing them throughout the article, where they become relevant.

Compiler. We implement a compiler that translates Choral source into Java libraries that comply with the source choreography: the code generated for each role performs the actions prescribed by the choreography. With our compiler, the programmer of the choreography is in control of the generated APIs: the APIs for each role follow the same structure found in the choreography, but where all data types and parameters in method signatures that do not pertain to the role are omitted, as we exemplified previously for the `authenticate` method.

Testing. We present the first testing tool for choreographic programming. Since choreography implementations are spread over multiple components (one for each role), testing choreographies can be difficult, because it calls for integration testing. Our testing tool leverages Choral to write

[1]We use a `Unit` return type—provided with Choral—instead of `void` for compositionality reasons, as discussed in Section 2.2.

'choreographic tests' that look like simple unit tests at the choreographic level, but are then compiled to integration tests that integrate the respective implementations of all roles.

Evaluation. We explore the expressivity of Choral with use cases of different kinds, covering security protocols, cyber-physical systems connected to the cloud, and parallel algorithms. For these use cases, we discuss relevant code excerpts and development techniques induced by Choral.

We then move from our examples to real-world comparisons.

First, we show how Choral can be used to transition existing Java programs to choreographies. We reimplement in Choral the Java implementation of Karatsuba's algorithm for fast multiplication, which yields a parallel implementation. Furthermore, we reimplement a clone of Twitter developed by the Spring team. Implementing this system in Choral requires identifying roles, which makes the original monolithic application much more modular.

Second, we compare Choral to a popular framework for concurrent programming in Java based on actors (Akka). In particular, we identify the key differences in the development processes and resulting architectures induced by the two technologies. We find that Choral provides concrete benefits in avoiding subtle concurrency bugs.

Third, we carry out a quantitative evaluation of how Choral impacts software development and runtime performance. Thanks to its choreographic approach, Choral consistently leads to smaller codebases. Moreover, the impact of our compiler on the speed of the edit-compile cycle is negligible (milliseconds). Finally, we show that the runtime performance of the code generated by Choral is not worse (and often better) than that of alternative implementations in Java and Akka.

Outline. We overview Choral with simple examples in Section 2, and give more realistic use cases in Section 3. The syntax and implementation of Choral are discussed in Section 4, and testing in Section 5. We evaluate Choral in depth in Section 6. Related and future work are discussed in Section 7. We draw conclusions in Section 8.

2 CHORAL IN PRACTICE

We start with an overview of the key features of Choral. First, spatial distribution: the expression of computation that takes place at different roles (Section 2.1). Second, interaction: the coding of data exchanges between roles (Section 2.2). Third, knowledge of choice: the coordination of roles to choose between alternative behaviours (Section 2.3).

The Choral language is quite big. Its usefulness depends on the capability to produce software libraries whose APIs look like "idiomatic" Java APIs, so we chose to incorporate a substantial set of features, which would commonly be considered necessary to use and produce Java APIs: Choral supports classes, interfaces, generics, inheritance, and method overloading. APIs generated by Choral support lambda expressions, in the sense that Java programmers can pass lambda expressions as arguments to our APIs. (Just as in Java, Choral sees these arguments as objects.) Supporting the Java syntax for lambda expressions inside Choral programs is not necessary for our objective, since they can be encoded as objects, so we leave it to future work on ergonomics.

In the rest of this section, we explain the key aspects of Choral by assuming that the reader is familiar with Java. The reader can assume that language constructs that have the same syntax as Java behave as expected (modulo our additions, which we explain in the text).

2.1 Roles and Data Types

Hello roles. All values in Choral are distributed over one or more roles, using the `@`-notation seen in Section 1. The degenerate case of values involving one role allows Choral to reuse existing Java classes and interfaces, lifted mechanically to Choral types and made available to Choral code. For example, the literal `"Hello from A"@A` is a string value `"Hello from A"` *located at role* `A`. Code involving different roles can be freely mixed in Choral, as in the following snippet.

```
1 class HelloRoles@(A,B) {
2  public static void sayHello() {
3   String@A a = "Hello from A"@A;
4   String@B b = "Hello from B"@B;
5   System@A.out.println(a);
6   System@B.out.println(b);
7  }
8 }
```
Choral Code

The code above defines a class, `HelloRoles`, parameterised over two roles, `A` and `B`. At line 3, we assign the string `"Hello from A"` located at `A` (`"Hello from A"@A`) to variable `a` of type "String at `A`" (`String@A`). At line 4, we do the same for a string located at `B`. Then, at line 5, we print variable `a` by using the `System` object at `A` (`System@A`); at line 6, we do the same for variable `b` at role `B`.

Roles are part of data types in Choral, adding a new dimension to typing. For example, the statement `String@A a = "Hello from B"@B` would be ill-typed, because the expression on the right returns data at a different role from that expected by the left-hand side.

Formally, in Choral, `String` is a type of a higher kind (or type constructor): it takes a role to return a type of the same kind of Java types [Moors et al. 2008]. The code `String@A` represents the instantiation of `String` at role `A`. Any Java type is automatically liftable to a Choral type with a single role parameter by following the same reasoning, enabling interoperability. Type constructors in Choral are not limited to a single role in general. We are going to see examples with multiple roles and more complex type parameters later in this section.

From Choral to Java. Given class `HelloRoles`, the Choral compiler generates for each role parameter a Java class with the behaviour for that role, in compliance with the source class. Here, the Java class for role `A` is `HelloRoles_A` and the class for `B` is `HelloRoles_B`.

```
class HelloRoles_A {
 public static void sayHello() {
  String a = "Hello from A";
  System.out.println(a);
 }
}
```
Generated Code

```
class HelloRoles_B {
 public static void sayHello() {
  String b = "Hello from B";
  System.out.println(b);
 }
}
```
Generated Code

Each generated class contains only the instructions that pertain to that role. If Java developers want to implement the behaviour of method `sayHello` for a specific role of the `HelloRoles` choreography, say `A`, then they just need to invoke the generated `sayHello` method in the respective generated class (`HelloRoles_A`). If all Java programs interested in participating in `HelloRoles` do that, then their resulting global behaviour complies by construction with the source choreography.

Notice that the code compiled for `A` and `B` will not interact and can therefore proceed fully concurrently, because the choreography does not prescribe so. In general, choreographic programming languages are expected to generate code that interacts only to enact the communications that the programmer specified in the choreography [Montesi 2023]. Choral follows this adequacy principle. We discuss how to program interactions in Section 2.2.

Distributed state. Fields of Choral classes carry state and can be distributed over different roles. For example, a class `DPair` can define a distributed structure storing two values at different roles.

```
1 class DPair@(A,B)<L@C,R@D> {
2  private L@A left;
3  private R@B right;
4  public DPair(L@A left, R@B right) { this.left = left; this.right = right; }
5  public L@A left() { return this.left; }
6  public R@B right() { return this.right; }
7 }
```

Choral Code

Class `DPair` is distributed between roles `A` and `B` and has two fields, `left` and `right`. In general, every class or interface in Choral is always parameterised on at least one role and, hence, it is a type constructor. The class is also parameterised on two data types, `L` and `R`, which exemplifies our support for generics [Naftalin and Wadler 2006]. At line 1, `L@C` specifies that `L` is expected to be a data type parameterised over a single role, abstracted by `C`; similarly for `R@D`. Naming the role parameters of `L` and `R` does not add any information in this particular example (we only need to know that they have one parameter). However, naming role parameters in generics is useful for expressing type bounds in **`extends`** clauses, as discussed later in this section. Choral interprets role parameter binders as in Java generics: the first appearance of a parameter is a binder, while subsequent appearances of the same parameter are bound. Observe that the scope of role parameters `C` and `D` is limited to the declaration of the type parameters `L` and `R`, respectively—we use distinct names exclusively for readability. At lines 2 and 3, we have the two fields, `left` and `right`, respectively, located at `A` and `B` as stated by the types `L@A` and `R@B`, the constructor is at line 4, while accessors to the two fields are at lines 5 and 6.

Data structures like `DPair` are useful when defining choreographies where the data at some role needs to correlate with data at another role, as with distributed authentication tokens. We apply them to a use case in Section 3.1.

2.2 Interaction

Choral programs become interesting when they contain interaction between roles—otherwise, they are simple interleavings of local independent behaviours by different roles, as in `HelloRoles`.

Choreography models typically come with some fixed primitives for interaction, e.g., sending a value from a role to another over a channel [Carbone et al. 2012]. Thanks to our data types parameterised over roles, Choral is more expressive: We can *program* these basic building blocks and then construct more complex interactions compositionally. This allows us to be specific about the requirements of choreographies regarding communications, leading to more reusable code. For instance, if a choreography needs only a directed channel, then our type system can see by subtyping that a bidirectional channel is also fine.

Directed data channels. We start our exploration of interaction in Choral from simple, directed channels for transporting data. In Choral, such a channel is just an object that takes data from one place to another. We can specify this behaviour as an interface.

```
interface DiDataChannel@(A,B)<T@C> {
 <S@D extends T@D> S@B com(S@A m);
}
```

Choral Code

A `DiDataChannel` is the interface of a directed channel between two roles, abstracted by `A` and `B`, for transmitting data of a given type, abstracted by the type parameter `T`, from `A` to `B` (hence the number of role parameters in `T`). Data transmission is performed by invoking the only method of the interface: `com`, which takes any value of a subtype of `T` located at `A`, `S@A`, and returns a value of type `S@B`. The type parameter `S` of method `com` has `T` as the upper bound (we can read the expression `S@D extends T@D` as "for any role `D`, `S@D extends T@D`") and allows us to transmit data of types more specific than `T` without losing type information (as it would be if the signature of `com` was simply `T@B com(T@A m)`).

Parameterising data channels over the type of transferrable data (`T`) is important in practice for channel implementors, because they often need to deal with data marshalling. Choral comes with a standard library that offers implementations of our channel APIs for a few common types of channels, e.g., TCP/IP sockets supporting JSON objects and shared memory channels. Users can provide their own implementations.

Using a `DiDataChannel`, we can write a simple method that sends a string notification from a `Client` to a `Server` and logs the reception by printing on screen.

```
notify(DiDataChannel@(Client,Server)<String> ch, String@Client msg) {
 String@Server m = ch.<String>com(msg);
 System@Server.out.println(m);
}
```
Choral Code

Note that `String` is a valid instantiation of `T` of `DiDataChannel`, because we lift all Java types as Choral types parameterised over a single role.

Alien data types. Compiling `DiDataChannel` to Java poses an important question: what should be the return type of method `com` in the code produced for role `A`? Since the return type does not mention `A` (we say that it is *alien* to `A`), a naïve answer to this question could be `void`, as follows.

```
interface DiDataChannel_A<T> {
 <S extends T> void com(S m);
}
```
Tentative Code

It turns out that this solution does not work well with expressions that compose multiple method calls, including chaining like `m1(e1,e2).m2(e3)` and nesting like `m1(m2(e))`. As a concrete example, consider a simple round-trip communication from `A` to `B` and back.

```
static <T@C> T@A roundTrip(DiDataChannel@(A,B)<T> chAB,DiDataChannel@(B,A)<T> chBA,T@A msg) {
 return chBA.<T>com(chAB.<T>com(msg));
}
```
Choral Code

Method `roundTrip` takes two channels, `chAB` and `chBA`, which are directed channels, respectively, from `A` to `B` and from `B` to `A`. The method sends the input `msg` from `A` to `B` and back by nested `com`s and returns the result at `A`.

A structure-preserving compilation of method `roundTrip` for role `A` would be as follows.

```
static <T> T roundTrip(DiDataChannel_A<T> chAB, DiDataChannel_B<T> chBA, T msg) {
 return chBA.<T>com(chAB.<T>com(msg));
}
```
Generated Code

Observe how the inner method call, `chAB.com<T>(msg)`, should return something, such that it can trigger the execution of the outer method call to receive the response. Therefore, the `com` method of `DiDataChannel_A` cannot have `void` as the return type.

Programming language experts have probably guessed by now that the solution is to use unit values instead of `void`. Indeed, Choral defines a singleton type `Unit` (a final class) that our compiler uses instead of `void` to obtain Java code whose structure resembles its Choral source code.

We now show the Java code produced by our compiler from `DiDataChannel` for both `A` and `B`.

```
interface DiDataChannel_A<T> {
 <S extends T> Unit com(S m);
}
```
Generated Code

```
interface DiDataChannel_B<T> {
 <S extends T> S com(Unit m);
}
```
Generated Code

Given these interfaces, the compilation of `roundTrip` for role `A` is well-typed and correct Java code. Without our usage of `Unit`, we would be forced to modify the structure of the compiled code wrt to its source. We chose in favour of our solution, because preserving structure makes it easier to read and debug the compiled code (especially when comparing it to the source choreography), and also makes our compiler simpler.

The users of our compiled libraries are not forced to pass `Unit` arguments to methods. Indeed, for methods like `com` of `DiDataChannel_B`, our compiler generates corresponding 'courtesy methods' that take no parameters and inject `Unit`s automatically.

Bidirectional channels. An immediate generalisation of directed data channels brings us to bidirectional data channels, specified by `BiDataChannel`.

```
interface BiDataChannel@(A,B)<T@C,R@D> extends DiDataChannel@(A,B)<T>,DiDataChannel@(B,A)<R>{
  <S@C extends T@C> S@B com(S@A m); // inherited from DiDataChannel@(A,B)<T>
  <S@C extends R@C> S@A com(S@B m); // inherited from DiDataChannel@(B,A)<R>
}
```
Choral Code

A `BiDataChannel` is parameterised over two types: `T` is the type of data that can be transferred from `A` to `B` and, vice versa, `R` is the type of data that can be transferred in the opposite direction. This is obtained by multiple type inheritance: `BiDataChannel` extends `DiDataChannel` in one and the other direction, which allows us to modularly use a bidirectional data channel in code that has the weaker requirement of a directed data channel in one of the two supported directions. Distinguishing the two parameters `T` and `R` is useful for protocols that have different types for requests and responses, like HTTP. Extending `DiDataChannel` twice does not result in any clashes, since `A` and `B` play different roles in each supertype. This "twin" inheritance results in the overload of method `com`, one for each communication direction supported by the channel. This overload does not result in any clash in the compilation, as illustrated by the code generated for `A` (the code generated for `B` is symmetric).

```
interface BiDataChannel_A<T,R> extends DiDataChannel_A<T>, DiDataChannel_B<R> {
  <S extends T> Unit com(S m); // inherited from DiDataChannel_A<T>
  <S extends R> S com(Unit m); // inherited from DiDataChannel_B<R>
}
```
Generated Code

We discuss more types of channels (including symmetric channels) in Section 2.4 and provide more details on inheritance and overloading in Section 4.

Forward chaining. We use bidirectional channels to define a choreography for remote procedure calls, called `RemoteFunction`, which leverages the standard Java interface `Function<T,R>`.

```
1  class RemoteFunction@(Client,Server)<T@A,R@B> {
2   private BiDataChannel@(Client,Server)<T,R> ch;
3   private Function@Server<T,R> f;
4   public RemoteFunction(BiDataChannel@(Client,Server)<T,R> ch, Function@Server<T,R> f) {
5    this.ch = ch; this.f = f;
6   }
7   public R@Client call(T@Client t) {
8    return ch.<R>com(f.apply(ch.<T>com(t)));
9   }
10 }
```
Choral Code

In our experience with programming larger Choral programs (as those in Section 3), we found it rather natural to compose method invocations that transfer data, as in line 8 of `RemoteFunction`. In these chains, we read data transfers from right to left (innermost to outermost invocation), but most choreography models in the literature use a left-to-right notation (as in "Alice sends 5 to Bob"). To make Choral closer to that familiar choreographic notation, we borrow the forward chaining operator `>>` from F# [Petricek and Skeet 2009], so that `exp >> obj::method` is syntactic sugar for `obj.method(exp)`. For example, we can rewrite method `call` of `RemoteFunction` as follows, which is arguably more readable and recovers a more familiar choreographic notation.

```
public R@Client call(T@Client t) {
 return t >> ch::<T>com >> f::apply >> ch::<R>com;
}
```
Choral Code

Using Choral libraries. As mentioned for Channels, when we compile the `RemoteFunction` class above, we obtain two Java classes: a `RemoteFunction_Client`, which sends some data to the `Server` for processing and waits for its response, and a `RemoteFunction_Server`, which, upon reception, feeds the data into a `Function` and sends back to the `Client` its result.

The `RemoteFunction_Server` is an interesting example of how users interact with Choral libraries. The code (snippet) generated from Choral is:

```
class RemoteFunction_Server<T,R> {
 private BiDataChannel_B <T,R> ch;
 private Function <T,R> f;
 public RemoteFunction_Server(BiDataChannel_B<T,R> ch, Function<T,R> f) { /*...*/ }
 public Unit call() { /*...*/ }
}
```
Generated Code

A user of the `RemoteFunction_Server` can interact in the choreography by providing the definition of the `Function` at the creation of the object. In general, this is how we expect users to integrate Choral-generated code with their "local code," i.e., code parametric to the choreography that users can implement locally, without any coordination with the other participants (save the APIs induced by Choral-generated code). For example, the snippet below is from a Java class that uses `RemoteFunction_Server` to provide a remote procedure for checking if an integer is even.

```
1 BiDataChannel_B<Integer,Boolean> channel = /*...*/;
2 new RemoteFunction_Server<Integer, Boolean>(channel, i -> i % 2 == 0).call();
```
Local Code

Here, at line 2 (second argument of the constructor), we provide the definition of the `Function` using Java Lambdas functional syntax.

2.3 Knowledge of Choice

Knowledge of choice is a hallmark challenge of choreographies: when a choreography chooses between two alternative behaviours, roles should coordinate to ensure that they agree on which behaviour should be implemented [Castagna et al. 2011].

We exemplify the challenge with the following code, which implements the consumption of a stream of items from a producer `A` to a consumer `B`.

```
1 // wrong implementation
2 consumeItems(DiDataChannel@(A,B)<Item@C> ch, Iterator@A<Item> it, Consumer@B<Item> consumer){
3  if (it.hasNext()) {
4   it.next() >> ch::<Item>com >> consumer::accept;
5   consumeItems(ch, it, consumer);
6  }
7 }
```
Choral Code

Method `consumeItems` takes a channel from `A` to `B`, an iterator over a collection of items at `A`, and a consumer function for items at `B`. Role `B` works reactively, where its consumer function is invoked whenever the stream of `A` produces an element: If the iterator can provide an item (line 3), then it is transmitted from `A` to `B`, consumed at `B`, and the method recurs to consume the other items (line 4).

The reader familiar with choreographies should recognise that this method implementation is *wrong*, due to (missing) knowledge of choice: The information on whether the if-branch should be entered or not is known only by `A` (since it evaluates the condition), so `B` does not know whether it should run lines 4 and 5 (receive, consume, and recur), or do nothing and terminate.

In choreographic programming, knowledge of choice is typically addressed by equipping the choreography language with a "selection" primitive to communicate constants drawn from a dedicated set of "labels" [Carbone and Montesi 2013; López et al. 2016]. This makes it possible for the compiler to build code that can react to choices made by other roles, inspired by a theoretical operator known as merging [Carbone et al. 2012]. In Choral, we adapt this practice to objects. Notably, Choral is expressive enough that we do not need to add a dedicated primitive or a dedicated set of labels.

We define a method-level annotation `@SelectionMethod`, which developers can apply only to methods that can transmit instances of enumerated types between roles (the compiler checks for this condition). For example, we can specify a directed channel for sending such enumerated values with the following `DiSelectChannel` interface.

```
interface DiSelectChannel@(A,B) {
 @SelectionMethod
 <T@C extends Enum@C<T>> T@B select(T@A m);
}
```
Choral Code

Our compiler assumes that implementations of methods annotated with `@SelectionMethod` return at the receiver the same value given at the sender. (This is part of the contract for channels, and it is a standard assumption in implementations of choreographies.)

Typically, channels used in choreographies are assumed to support both data communications and selections. We can capture this functionality with `DiChannel` (directed channel), a subtype of both `DiDataChannel` and `DiSelectChannel` (we include inherited methods for convenience).

```
interface DiChannel@(A,B)<T@C> extends DiDataChannel@(A,B)<T>, DiSelectChannel@(A,B) {
  <S@C extends T@C> S@B com(S@A m);          // inherited from DiDataChannel@(A,B)
  <S@C extends Enum@C<T>> T@B select(T@A m); // inherited from DiSelectChannel@(A,B)
}
```
Choral Code

Using `DiChannels`, we can update `consumeItems` to respect knowledge of choice.

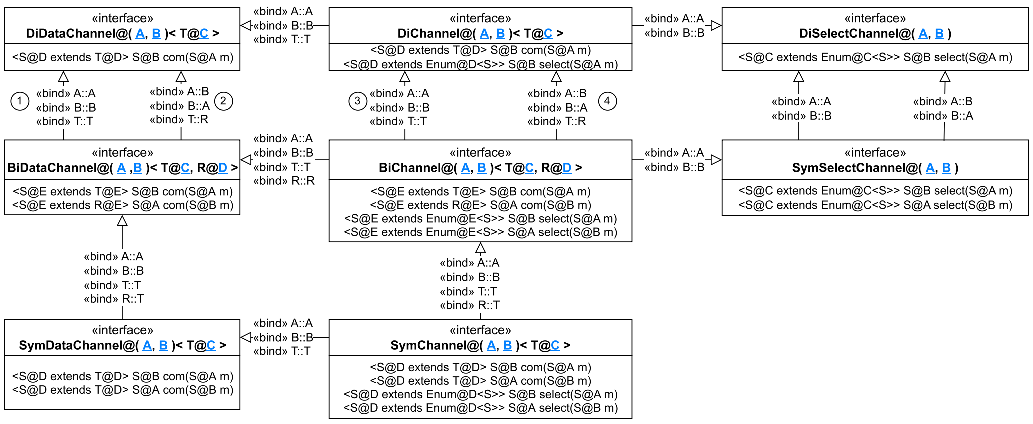

«interface»
DiDataChannel@(A, B)< T@C >
<S@D extends T@D> S@B com(S@A m)
«bind» A::A
«bind» B::B
«bind» T::T
«interface»
DiChannel@(A, B)< T@C >
<S@D extends T@D> S@B com(S@A m)
<S@D extends Enum@D<S>> S@B select(S@A m)
«bind» A::A
«bind» B::B
«interface»
DiSelectChannel@(A, B)
<S@C extends Enum@C<S>> S@B select(S@A m)
1
«bind» A::A
«bind» B::B
«bind» T::T
«bind» A::B
«bind» B::A
«bind» T::R
2
3
«bind» A::A
«bind» B::B
«bind» T::T
«bind» A::B
«bind» B::A
«bind» T::R
4
«bind» A::A
«bind» B::B
«bind» A::B
«bind» B::A
«interface»
BiDataChannel@(A ,B)< T@C, R@D >
<S@E extends T@E> S@B com(S@A m)
<S@E extends R@E> S@A com(S@B m)
«bind» A::A
«bind» B::B
«bind» T::T
«bind» R::R
«interface»
BiChannel@(A, B)< T@C, R@D >
<S@E extends T@E> S@B com(S@A m)
<S@E extends R@E> S@A com(S@B m)
<S@E extends Enum@E<S>> S@B select(S@A m)
<S@E extends Enum@E<S>> S@A select(S@B m)
«bind» A::A
«bind» B::B
«interface»
SymSelectChannel@(A, B)
<S@C extends Enum@C<S>> S@B select(S@A m)
<S@C extends Enum@C<S>> S@A select(S@B m)
«bind» A::A
«bind» B::B
«bind» T::T
«bind» R::T
«bind» A::A
«bind» B::B
«bind» T::T
«bind» R::T
«interface»
SymDataChannel@(A, B)< T@C >
<S@D extends T@D> S@B com(S@A m)
<S@D extends T@D> S@A com(S@B m)
«bind» A::A
«bind» B::B
«bind» T::T
«interface»
SymChannel@(A, B)< T@C >
<S@D extends T@D> S@B com(S@A m)
<S@D extends T@D> S@A com(S@B m)
<S@D extends Enum@D<S>> S@B select(S@A m)
<S@D extends Enum@D<S>> S@A select(S@B m)

Fig. 3. UML class diagram of the hierarchy of the *Channel interfaces.

```
enum Choice@A { GO, STOP }
```
Choral Code

```
consumeItems(DiChannel@(A,B)<Item> ch, Iterator@A<Item> it, Consumer@B<Item> consumer) {
 if (it.hasNext()) {
  ch.<Choice>select(Choice@A.GO);
  it.next() >> ch::<Item>com >> consumer::accept;
  consumeItems(ch, it, consumer);
 } else {
  ch.<Choice>select(Choice@A.STOP);
 }
}
```
Choral Code

Differently from the previous, broken implementation of `consumeItems`, now role `A` sends a selection of either `GO` or `STOP` to `B`. Role `B` can now inspect the received enumerated value to infer whether it should execute the code for the if- or the else-branch of the conditional. This information is exploited by our static analyser to check that `consumeItems` respects knowledge of choice, and also by our compiler to generate code for `B` that reacts correctly to the choice performed by `A`. (We provide a more extensive example, including the code compiled for the receiver, in Section 3.1.)

Our compiler supports three features to make knowledge of choice flexible. First, our check for knowledge of choice works with arbitrarily nested conditionals. Second, knowledge of choice can be propagated transitively. Say that a role `A` makes a choice that determines that two other roles `B` and `C` should behave differently, and `A` informs `B` of the choice through a selection. Now either `A` or `B` can inform `C` with a selection, because our compiler sees that `B` now possesses knowledge of choice. Third, knowledge of choice is required only when necessary: If `A` makes a choice and another role, say `B`, does not need to know, because it performs the same actions (e.g., receiving an integer from `A`) in both branches, then no selection is necessary. We explain the technicalities of this check in Section 4.

2.4 The Family of Choral Channels

Choral types give us a new way to specify requirements on channels that prior work implicitly assumed, leading to the definition of a family of channel interfaces diagrammed in Figure 3.

From the left-most column in Figure 3, at the top, we find `DiDataChannel`, representing a directed channel parameterised over `T` (the type of data that can be sent). We obtain `BiDataChannel`, a bidirectional data channel, by extending `DiDataChannel` once for each direction: ① it binds the role parameters of one extension in the same order given for the role parameters of `BiDataChannel`, giving us a direction from `A` to `B` and ② it binds the role parameters of the other extension in the opposite way, giving us a direction from `B` to `A`. The result is that `BiDataChannel` defines two `com` methods: one transmitting from `A` to `B`, the other from `B` to `A`. The last lines in ① and ② in Figure 3 complete the picture: the first generic data type `T` binds data from `A` to `B`, the second generic data type `R` binds data from `B` to `A`. The `SymDataChannel` in Figure 3, by extending the `BiDataChannel` interface and binding the two generic data types `T` and `R` with its only generic data type `T`, defines a bidirectional data channel that transmits one type of data, regardless of its direction.

The right-most vertical hierarchy in Figure 3 represents channels supporting selections and it follows a structure similar to that of data channels. A `DiSelectChannel` is a directed selection channel and a `SymSelectChannel` is the bidirectional version—there is no `BiSelectChannel` since both directions exchange the same enumerated types.

The vertical hierarchy in the middle column of Figure 3 is the combination of the left-most and right-most columns. Interface `DiChannel` is a directed channel that supports both generic data communications and selections. `BiChannel` is its bidirectional extension (③ and ④ in Figure 3), and `SymChannel` is the symmetric extension of `BiChannel`. The snippet below contains the definition of the interface `BiChannel`.

```
interface BiChannel@(A,B)<T@C, R@D> extends
    DiChannel@(A,B)<T>,         // A BiChannel is a pair of directed channels
    DiChannel@(B,A)<R>,         // in opposite directions
    BiDataChannel@(A,B)<T,R>,   // that supports data
    BiSelectChannel@(A,B)       // and selections
  { } // we do not define any new methods, since they are all inherited
```
Choral Code

This definition means that for any pair of distinct roles `C`, `D` and for any pair of types `S`, `P` (with one role parameter), `BiChannel@(C,D)<S,P>` is a subtype of `DiChannel@(C,D)<S>`, `DiChannel@(D,C)<P>`, `BiDataChannel@(C,D)<S,P>`, and `BiSelectChannel@(C,D)`.

Implementing Choral channels. Our channel interfaces can be implemented directly in Choral or in Java. We exemplify the latter case, which lets us also show how one can carry out this task by fully leveraging the existing Java language and ecosystem.

Let us say we want to implement a symmetric channel over TCP/IP that a client (say `A`) is going to use for transmitting strings to a server (say `B`). We can implement this channel by writing a class `TCPClientChannel` that implements the interface generated for `A` from `SymChannel` (instantiating its generic with `String`). The snippet below shows the structure of such an implementation.

```
public class TCPClientChannel implements SymChannel_A<String> {
 private final SocketChannel socketChannel;
 private TCPClientChannel(SocketChannel socketChannel) {
  this.socketChannel = socketChannel;
 }
 public void com(String m) { /*...*/ }
 public String com() { /*...*/ }
 /*...*/
 public static TCPClientChannel open(SocketAddress remote) throws IOException {
  return new TCPClientChannel(SocketChannel.open(remote));
 }
}
```

The `TCPClientChannel` uses `SocketChannel` from the Java standard library. The class also offers a factory method for connecting to a remote address in the standard way. This method creates a channel that the user can pass as an argument to the code generated from a choreography. In particular, the `TCPClientChannel` is designed to provide a conventional socket set-up experience for Java programmer, where the expected contract relies on opening a connection based on a `SocketAddress`. Implementing the server-side counterpart of `TCPClientChannel` is straightforward and follows the same approach.

2.5 Handling Exceptions

Typically, choreographic languages assume reliable communications [Carbone et al. 2008; Carbone and Montesi 2013; Dalla Preda et al. 2017]. The only exception is the language theory in Montesi and Peressotti [2017], which shows that one can relax this assumption, by allowing the choreographic language to handle local exceptions. In Choral, we follow the same strategy, which we briefly illustrate here.

Choral can invoke Java code, which might raise an exception. Plain Java code is always located at one role, and therefore the same holds for exceptions (exceptions are "local"). We exemplify how we treat exceptions with the following choreography, where a role `B` uses the Java standard library to save on disk some text communicated from another role `A`.

```
 1  public Result@A<String, String> save(
 2   SymDataChannel@(A,B)<String> chAB, String@A text, Path@B path
 3  ) {
 4   String@B textB = text >> chAB::<String>com;
 5   Result@B<String, String> result;
 6   try {
 7    Files@B.writeString(path, textB);
 8    result = Result@B.ok("Saved"@B);
 9   } catch(IOException@B ex) {
10    result = Result@B.err(ex.getMessage());
11   }
12   return result >> chAB::<String>com;
13  }
```
Choral Code

Above, we start by communicating the text to be saved from `A` to `B` (line 4). We then declare at `B` a `result` variable (line 5), which will store either a success or error message—that `B` later communicates to `A` (line 12). At line 7, `B` attempts to save the received text.

This choreography might incur execution errors related to communication or file writing. Exceptions encapsulate these errors. For example, the invocation of method `writeString` might throw an `IOException` located at `B`, which we handle with the **`try-catch`** block at lines 6–11.

Method `com` can throw exceptions too, depending on the implementation of channel `ch`. Channels for remote communication (e.g., based on TCP/IP sockets) in the Choral library use the following strategy: the sender attempts at sending a message until its network stack accepts the task (using exponential backoff and bound by a maximal number of attempts, to guarantee termination); likewise, the receiver attempts at receiving until a timeout expires. If the sender ultimately fails at relaying the message to its local network stack, then the channel throws a `SendException` at the sender (`A` in our example). If the receiver timeouts before receiving a message, then the channel raises a `TimeoutException` at the receiver (`B` in our example).

As a design choice, we left the exceptions of method `com` unchecked. The idea is that the implementation of channels should do their best to deliver messages, and when this is not possible the local code that uses the code generated from a choreography should deem the execution of the choreography unsuccessful. The local code is free to catch these exceptions and attempt recovery, for example by executing the choreography again (as in actor frameworks [Wyatt 2013]).

However, our implementation of `com` in the Choral standard library is just a default; Choral does not hardcode any communication semantics. The user is free to implement alternative communication methods that expose an API, which the caller choreography might use to handle network errors, e.g., lost messages. For instance, we might account for lossy communications between A and B within the above choreography as follows.

```
public Result@A<String, String> save(
 LossySymDataChannel@(A,B)<String> chAB, String@A text, Path@B path
) {
 Optional@B<String> textB = text >> chAB::<String>lossyCom;
 Result@B<String, String> result;
 if(!textB.isEmpty()) {
  try {
   Files@B.writeString(path, textB.get());
   result = Result@B.ok("Saved"@B);
  } catch(IOException@B ex) {
   result = Result@B.err(ex.getMessage());
  }
 } else {
  result = Result@B.err("Network error");
 }
 return result >> chAB::<String>com;
}
```
Choral Code

Channel `chAB` is now a `LossySymDataChannel` and, in addition to method `com`, it offers also method `lossyCom`. The latter does not throw exceptions in case of communication failures but rather returns an `Optional` value that contains the received value in case of success or is empty otherwise.

Choosing which errors a choreography should deal with and which errors should be raised as unrecoverable exceptions to the local code is a design trade-off that derives from the usual tension between robustness versus simplicity. This trade-off is typical of coordination protocols and exists independently from Choral, which is why we decided to leave the programmer free to navigate this spectrum. Gathering from our own experience with Choral, we recognised the following design principles. Protocols whose design assumes a reliable network layer should not deal with communication errors within the choreography (e.g., the Diffie-Hellman protocol for key exchange, which we briefly describe below and that we implemented for our evaluation in Section 6). Contrarily, choreographies implementing protocols designed to deal with network errors should specify the handling of those errors (e.g., implementations of objects dedicated to data transfer, like our channels). Choral is quite flexible regarding these aspects. A channel API can offer methods that both raise exceptions—like method `com`, meaning that the communication is essential—or wrap failures in data types (raising no exceptions)—like method `lossyCom`, meaning that the communication is not essential and that the choreography can handle internally the failure. The programmer can use the different methods within the same choreography to pinpoint which communications are deemed essential and which are not.

2.6 What Goes in a Choreography?

We have just looked at the design issue of deciding whether to deal with errors in the choreography or in the local code that uses the (communication code compiled from the) choreography. This is an instance of the more general issue of protocol design: what should be part of a choreography? This issue exists even when designing choreographies informally (without Choral), because one needs to choose what details are fixed in the protocol and what is instead left to the discretion of the local code. There is no one-size-fits-all solution, since these choices are influenced by the concrete use case that the choreography deals with.

Consider, for example, the widely adopted Diffie-Hellman protocol for cryptographic key exchange [Diffie and Hellman 1976]. Integral parts of the protocol specify both computation and communication. In the protocol, two parties, e.g., `Alice` and `Bob`, use two pairs of keys (a private and a public one) to generate a shared secret, which they can later use for symmetric encryption. Formally, let p be a prime number and g be a primitive root modulo p, sA be a secret key held by `Alice`, and sB be a secret key held by `Bob`. First, `Alice` computes her public key $pA = (g^{sA} \bmod p)$ and, likewise, `Bob` computes his public key $pB = (g^{sB} \bmod p)$. Then, `Alice` and `Bob` exchange their public keys, which they can use to generate their shared secret s:

$$\overbrace{\underbrace{(g^{sB} \bmod p)}_{\texttt{Bob}\text{'s } pB}{}^{sA} \bmod p}^{\texttt{Alice}\text{'s side}} = \underbrace{s}_{\text{shared secret}} = \overbrace{\underbrace{(g^{sA} \bmod p)}_{\texttt{Alice}\text{'s } pA}{}^{sB} \bmod p}^{\texttt{Bob}\text{'s side}}.$$

Since the computations performed by `Alice` and `Bob` are essential to the protocol, any faithful Choral implementation shall include those details too: doing otherwise would mean implementing a different protocol. The following is a snippet of the implementation that we have written for our evaluation in Section 6 (with variables renamed to match our description above).

```
BigInteger@Bob pA = g.modPow(sA, p) >> channel::<BigInteger>com;
BigInteger@Bob s = pA.modPow(sB, p);
```
Choral Code

Differently from these computational details, the Diffie-Hellman protocol does not fix the implementation of the channel used to communicate data. It is therefore reasonable that a Choral implementation of the protocol is parameterised on this implementation—we reflected this condition by having `channel` as a parameter of the method that contains our code above.

An example of a choreography where the definition of computation is completely abstracted away is the `consumeItems` method in Section 2.3. The choreography fixes the coordination between the participants, but not how they produce or consume the data to be exchanged. The latter is to be defined by either local code or another choreography that invokes `consumeItems`.

In general, how much computation should be defined in a choreography forms a spectrum. A "good" choreographic programming language should thus give freedom to define or abstract away computation at will. Choral provides this capability through the standard facilities of object-oriented programming (parameters, inheritance, etc.).

3 USE CASES

We dedicate this section to illustrating how the features of Choral contribute to writing realistic choreographies. We start with a protocol for distributed authentication (Section 3.1), which we then reuse modularly in another use case from the healthcare sector that mixes cloud computing, edge computing, and Internet of Things (IoT) (Section 3.2). Finally, we show a use case on parallel computing, by showing a distributed implementation of merge sort (Section 3.3).

3.1 Distributed Authentication

We write a choreography for distributed authentication, inspired by the single sign-on authentication scheme: an `IP` ("Identity Provider," also known as central authentication service) authenticates a `Client` that accesses a third-party `Service`.

We start by introducing an auxiliary class, `AuthResult`, that we will use to store the result of authentication. The idea is that, after performing the authentication protocol, both the `Client` and the `Server` should have an authentication token if the authentication succeeded, or an "empty" value if it failed. We model this behaviour by extending the `DPair` class presented in Section 2.

```
public class AuthResult@(A,B)
  extends DPair@(A,B)<Optional@A<AuthToken>,Optional@B<AuthToken>> {
 public AuthResult(AuthToken@A t1, AuthToken@B t2) {
  super(Optional@A.<AuthToken>of(t1), Optional@B.<AuthToken>of(t2));
 }
 public AuthResult() {
  super(Optional@A.<AuthToken>empty(), Optional@B.<AuthToken>empty());
 }
}
```
Choral Code

The constructors of `AuthResult` guarantee that either both roles (`A` and `B`) have an optional containing a value or both optionals are empty (`Optional` is the standard Java type). Since `AuthResult` extends `DPair`, these values are locally available by invoking the `left` and `right` methods.

We now present the choreography for distributed authentication, as the `DistAuth` class below.

```
enum AuthBranch { OK, KO }

public class DistAuth@(Client,Service,IP){
 private TLSChannel@(Client,IP)<Object> ch_Client_IP;
 private TLSChannel@(Service,IP)<Object> ch_Service_IP;

 public DistAuth(
  TLSChannel@(Client,IP)<Object> ch_Client_IP,
  TLSChannel@(Service,IP)<Object> ch_Service_IP ) {
  this.ch_Client_IP = ch_Client_IP;
  this.ch_Service_IP = ch_Service_IP;
 }

 private String@Client calcHash(String@Client salt, String@Client pwd) { /*...*/ }

 public AuthResult@(Client,Service) authenticate(Credentials@Client credentials) {
  String@Client salt = credentials.username
   >> ch_Client_IP::<String>com >> ClientRegistry@IP::getSalt >> ch_Client_IP::<String>com;
  Boolean@IP valid = calcHash(salt, credentials.password)
   >> ch_Client_IP::<String>com >> ClientRegistry@IP::check;
  if (valid) {
   ch_Client_IP.<AuthBranch>select(AuthBranch@IP.OK);
   ch_Service_IP.<AuthBranch>select(AuthBranch@IP.OK);
   AuthToken@IP t = AuthToken@IP.create();
25     return new AuthResult@(Client,Service)(
26      ch_Client_IP.<AuthToken>com(t), ch_Service_IP.<AuthToken>com(t)
27     );
28    } else {
29     ch_Client_IP.<AuthBranch>select(AuthBranch@IP.KO);
30     ch_Service_IP.<AuthBranch>select(AuthBranch@IP.KO);
31     return new AuthResult@(Client,Service)();
32    }
33   }
34  }
```

Choral Code

Class DistAuth is a *multiparty* protocol parameterised over three roles: Client, Service, and IP (for Identity Provider). It composes two channels as fields (lines 4 and 5), which, respectively, connect Client to IP and Service to IP—hence, the interaction between Client and Service can only happen if coordinated by IP. The channels are of type TLSChannel, a class for secure channels from the Choral standard library that uses TLS for security and the Kryo library [Grotzke 2020] for marshalling and unmarshalling objects. Class TLSChannel implements interface SymChannel, from Section 2, so it can be used in both directions. The private method calcHash (omitted) implements the local code that Client uses to hash its password.

Method authenticate (lines 16–33) is the key piece of DistAuth, which implements the authentication protocol. It consists of three phases. In the first phase, lines 17 and 18, the Client communicates its username to IP, which IP uses to retrieve the corresponding salt in its local database ClientRegistry; the salt is then sent back to Client. The second phase (lines 19 and 20) deals with the resolution of the authentication challenge. Client computes its hash with the received salt and its locally stored password and sends the latter to IP. IP then checks whether the received hash is valid, storing this information in its local variable valid. The result of the check is a Boolean stored in the valid variable located at IP. The first two phases codify some best practices for distributed authentication and password storage [Grassi et al. 2017]: the identity provider IP never sees the password of the client, but only its attempts at solving the challenge (the salt), which Client can produce with private information (here, its password). In the third phase, (lines 21–32), IP decides whether the authentication was successful or not by checking valid. In both cases, IP informs the Client and the Service of its decision, using selections to distinguish between success (represented by OK) or failure (represented by KO). In case of success, IP creates a new authentication token (line 24) and communicates the token to both Client and Service (inner calls to com at line 26). The protocol can now terminate and return a distributed pair (an AuthResult) that stores the same token at both Client and Service, which they can use later for further interactions (line 25). In case of failure, the method returns an authentication result with empty Optionals (line 31).

New to choreographic programming, DistAuth is a higher-order choreography: the channels that it composes are choreographies for secure communication that carry state—the result of the TLS handshake, which method com of TLSChannel uses internally. Taking this even further, we could overload method authenticate with a continuation-passing style alternative that, instead of returning a result, takes as parameters choreographic continuations (objects that involve Client and Service) to be called, respectively, in case of success (line 25) or failure (line 31).

Compilation. We now discuss key parts of the compilation of DistAuth for role Client, i.e., the Java library that clients can use to authenticate to an identity provider and access a service.

```
1  public class DistAuth_Client {
2   private TLSChannel_A<Object> ch_Client_IP;
3
4   public DistAuth_Client(TLSChannel_A <Object> ch_Client_IP) {
5    this.ch_Client_IP = ch_Client_IP;
6   }
7
8   private String calcHash( String salt, String pwd ) { /*...*/ }
9
10  public AuthResult_A authenticate(Credentials credentials) {
11   String salt = ch_Client_IP.<String>com(ch_Client_IP.<String>com(credentials.username));
12   ch_Client_IP.<String>com(calcHash(salt, credentials.password));
13   switch (ch_Client_IP.<AuthBranch>select(Unit.id)) {
14    case OK -> { return new AuthResult_A( ch_Client_IP.<AuthToken>com(Unit.id), Unit.id); }
15    case KO -> { return new AuthResult_A(); }
16    default -> { throw new RuntimeException( /*...*/ ); }
17   }
18  }
19 }
```
Generated Code

The field, constructor, and method at lines 2–8 are straightforward projections of the source class for role `Client`—fields and parameters pertaining only to other roles disappeared. The interesting code is at lines 10–17, which defines the local behaviour of `Client` in the authentication protocol. Note that forward-chaining sequences (`>>`) become plain nested calls in Java (lines 11 and 12). At line 11, the client sends its username to the identity provider and receives back the `salt`. Recall from Section 2 that the innermost invocation of method `com` returns a `Unit` since the client acts as the sender here. Once the username is sent, the innermost `com` returns, and we run the outermost invocation of `com`, which received the salt through the channel with the identity provider. At line 12, the Client sends the computed hash to the identity provider.

At line 13, we see an example of how our compiler implements knowledge of choice for roles that need to react to decisions made by other roles. The client receives an enumerated value of type `AuthBranch`, which can be either `OK` or `KO`, through the channel with the identity provider. Then, a `switch` statement matches the received value to decide whether (case `OK`) we shall receive an authentication token from the identity provider and store it as an `AuthResult_A` or (case `KO`) the authentication procedure failed.

3.2 A Use Case from Healthcare: Handling Streams of Sensitive Vitals Data

In this use case, we exemplify how developers can locally compose the libraries generated by independent choreographies, using a healthcare use case inspired by previous works on edge computing and pseudonymisation [Giallorenzo et al. 2019; Swaroop et al. 2019].

Suppose that a "healthcare service" in a hospital needs to gather sensitive data about vital signs (we call them vitals) from some IoT devices (e.g., smartwatches, heart monitors), and then upload them to the cloud for storage. This is a typical scenario that requires the integration of libraries for participating in choreographies at the local level. We shall carry out the following two steps.

(1) Define a new choreography class, called `VitalsStreaming`, that prescribes how data should be streamed from an IoT `Device` monitoring the vitals of a patient to a data `Gatherer`; this choreography shall enforce that the `Gatherer` processes only data that is (a) correctly cryptographically signed by the device and (b) pseudonymised.

(2) Implement the healthcare service as a local Java class, called `HealthCareService`, that combines the Java library compiled from `VitalsStreaming` to gather data from the IoT devices with the Java library compiled from our previous `DistAuth` example, to authenticate at the cloud storage service through a third-party service (this could be, e.g., a national authentication system) and upload the data.

Vitals choreography. `VitalsStreaming` implements the choreography for streaming vitals.

```
 1  public enum StreamState@R { ON, OFF }
 2
 3  public class VitalsStreaming@(Device,Gatherer) {
 4   private SymChannel@(Device,Gatherer)<Object> ch;
 5   private Sensor@Device sensor;
 6
 7   public VitalsStreaming(SymChannel@(Device,Gatherer)<Object> ch, Sensor@Device sensor) {
 8    this.ch = ch;
 9    this.sensor = sensor;
10   }
11
12   private static Vitals@Gatherer pseudonymise(Vitals@Gatherer vitals) { /*...*/ }
13   private static Boolean@Gatherer checkSignature(Signature@Gatherer signature) { /*...*/ }
14
15   public void gather(Consumer@Gatherer<Vitals> consumer) {
16    if (sensor.isOn()) {
17     ch.<StreamState>select(StreamState@Device.ON);
18     VitalsMsg@Gatherer msgOpt = sensor.next() >> ch::<VitalsMsg>com;
19     if (checkSignature(msg.signature())) {
20      msg.content() >> this::pseudonymise >> consumer::accept;
21     }
22     gather(consumer);
23    } else {
24     ch.<StreamState>select(StreamState@Device.OFF);
25    }
26   }
27  }
```
Choral Code

At lines 3–5, the class `VitalsStreaming` composes a channel between the `Device` and the `Gatherer` and a `Sensor` object located at the `Device` (for obtaining the local vital readings). At line 12, we define a method that pseudonymises personal data in `Vitals` at the `Gatherer`. Likewise, at line 13, we have a method that the `Gatherer` uses to check that a message signature is valid. (We omit the bodies of these two static methods, which are standard local methods.) The interesting part of this class is method `gather` (lines 15–26). The `Device` checks whether its sensor is on (line 16) and informs the `Gatherer` of the result with appropriate selections for knowledge of choice (lines 17 and 24). If the sensor is on, then `Device` sends its next available reading to `Gatherer` (line 18). `Gatherer` now checks that the message is signed correctly (line 19); if so, it pseudonymises the content of the message and then hands it off to a local consumer function. Notice that `Gatherer` does not need to inform `Device` of its local choice, since it does not affect the code that `Device` needs to run. We then recursively invoke `gather` to process the next reading.

Local code of the healthcare service. The local implementation of the healthcare service acts as `Gatherer` in the `VitalsStreaming` choreography (to gather the data) and as the `Client` in the `DistAuth` choreography (to authenticate with the cloud storage). So, we compose the compiled Java classes `VitalsStreaming_Gatherer` and `DistAuth_Client`, respectively.

```
1  public class HealthCareService {
2   public static void main() {
3    TLSChannel_A toIP = HealthIdentityProvider.connect();
4    MQTTClient toStorage = HealthDataStorage.connect();
5    AuthResult_A authResult = new DistAuth_Client(toIP).authenticate(getCredentials());
6    authResult.left().ifPresent(token -> {
7     DeviceRegistry
8      .parallelStream()
9      .forEach(device ->
10      Supervision.restart(() ->
11       new VitalsStreaming_Gatherer(device.connect())
12        .gather(data -> toStorage.com(new StorageMesg(token, data)))
13      )
14     );
15   });
16  }
17  private static Credentials getCredentials() { /* ... */ }
18 }
```

Local Code

Above, the `main` method idiomatically combines Java standard libraries with those generated by our compiler. At lines 3 and 4, we use auxiliary methods to connect to the identity provider (which implements `IP` in `DistAuth`) and the data storage service (which implements `Service` in `DistAuth`)—these services are provided by third parties, e.g., the national health system and some cloud provider. We choose a TLS channel to enact authentication. Instead, for communications with devices and storage, we use the MQTT protocol, which is typical for IoT applications [Hunkeler et al. 2008]—`MQTTClient` implements (the projection of) interface `SymChannel`, dealing with possible connectivity issues. At line 5, we run our distributed-authentication protocol as the `Client`. At line 6, we check if we successfully received an authentication token by inspecting the optional result. If so, then we obtain a parallel stream of `Device` objects from a local registry (lines 7 and 8). Each device (line 9) is handled by a worker that uses a restart supervision strategy (line 10), i.e., if the projection of the choreography `VitalsStreaming` encounters an unrecoverable error, then it is restarted. At line 11, we create a new instance of `VitalsStreaming_Gatherer` (the code compiled for `Gatherer` from `VitalsStreaming`), which receives an MQTT channel for communicating with the device (obtained by `device.connect()`). Finally, at line 12, we call the `gather` method to engage in the `VitalsStreaming` choreography with each device, passing a consumer function that sends the received data to the cloud storage service (including the authentication token).

Notice that we do not need to worry about pseudonymisation or signature checking in the local code, since the code compiled from `VitalsStreaming` manages all these details.

3.3 Merge Sort

The last use case that we present is a three-way concurrent implementation of merge sort [Knuth 1998], which illustrates the design of parallel algorithms in Choral. It also serves to showcase how role instantiation can help Choral programmers with writing load-distribution logic.

We define a class `MergeSort` parameterised on three roles: the *merger* (`M`), which is the participant that holds the list of elements that have to be sorted, and two *sorters* (`S1`, `S2`). In the code of `MergeSort`, the merger splits its list into two halves and, respectively, communicates them to the two sorters. The sorters are responsible for sorting their sublists and communicating the results back to the merger. When the merger gets the ordered sublists, it merges them in the standard way to compute the complete ordered list. The definition of `MergeSort` follows.

```
 1 enum Choice@R { L, R }
 2 public class Mergesort@(M,S1,S2){
 3  SymChannel@(M,S1)<Object> ch_MS1;
 4  SymChannel@(S1,S2)<Object> ch_S1S2;
 5  SymChannel@(S2,M)<Object> ch_MS2;
 6 
 7  public MergeSort(
 8   SymChannel@(M,S1)<Object> ch_MS1,
 9   SymChannel@(S1,S2)<Object> ch_S1S2,
10   SymChannel@(S2,M)<Object> ch_MS2 ) {
11   this.ch_MS1 = ch_MS1; this.ch_S1S2 = ch_S1S2; this.ch_MS2 = ch_MS2;
12  }
13 
14  public List@M<Integer> sort(List@M<Integer> a) {
15   if (a.size() > 1@M) {
16    ch_MS1.<Choice>select(Choice@M.L);
17    ch_MS2.<Choice>select(Choice@M.L);
18    Double@M pivot = a.size() / 2@M >> Math@M::floor >> Double@M::valueOf;
19    MergeSort@(S1,S2,M) mb = new MergeSort@(S1,S2,M)(ch_S1S2, ch_MS2, ch_MS1);
20    MergeSort@(S2,M,S1) mc = new MergeSort@(S2,M,S1)(ch_MS2, ch_MS1, ch_S1S2);
21    List@S1<Integer> lhs = a.subList(0@M,pivot.intValue())
22     >> ch_MS1::<List<Integer>>com >> mb::sort;
23    List@S2<Integer> rhs = a.subList(pivot.intValue(), a.size())
24     >> ch_MS2::<List<Integer>>com >> mc::sort;
25    return merge(lhs >> ch_MS1::<List<Integer>>com, rhs >> ch_MS2::<List<Integer>>com);
26   } else {
27    ch_MS1.<Choice>select(Choice@M.R);
28    ch_MS2.<Choice>select(Choice@M.R);
29    return a;
30   }
31  }
32 
33  private List@M<Integer> merge(List@M<Integer> lhs, List@M<Integer> rhs) { /* ... */ }
34 }
```
Choral Code

In the code, the interaction logic that we have just described is implemented in method `sort`. Method `merge`, instead, merges two lists at `M`; we omit it, since it is entirely local and works as in the standard sequential algorithm. Method `sort` consists of a conditional that checks whether the list should be split, which is determined by having more than one element (line 15). If so, then the merger finds a pivot (line 18) and uses it to split the list and communicate the resulting sublists to the two sorters (lines 21–24). The interesting part is that we recursively instantiate (lines 19 and 20) and use `MergeSort` (lines 22 and 24) to accomplish the two subtasks given to the sorters. In these recursive invocations, roles are switched to put each sorter in charge of its sublist: the sorter acts as the merger, and the other two nodes act as sorters. This goes on until the list to be sorted cannot be split anymore. The ordered sublists computed by each sorter are then merged at line 25.

The sequence diagram in Figure 4 exemplifies the coordination pattern codified by our choreography (for brevity, we omit selections). In the diagram, we have three endpoint nodes—Node$_1$, Node$_2$, and Node$_3$—which engage in the choreography by playing the respective roles `M`, `S1`, and

S2. We use numbered subscripts to denote the round (recursive step in the algorithm) that each interaction belongs to (sort_1, sort_2). The initial input list is [15, 3, 14]; it is located at Node_1, which in the beginning plays role M. In the first round (first invocation of `sort`), Node_1 asks Node_2 and Node_3 to sort the sublists obtained from the initial list—respectively, [15, 3] and [14]. This action starts a recursive call (second round) where Node_2 is the merger (M) and the other two nodes play the sorters. The lists communicated to the sorters in this round are [15] and [3]. These lists contain only one element each, so they are dealt with by the else-branch of the conditional in method `sort` and we do not start other rounds. Node_2 now collects the lists from Node_1 and Node_2 and merges them locally, obtaining [3, 15].

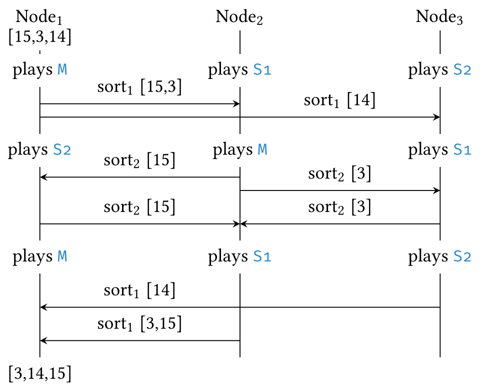


Fig. 4. Sequence diagram of data exchanges in the three-way distributed merge sort (selections are omitted).

Now that the recursive invocation for sorting [15, 3] is complete, we are back to completing the invocation of `sort` where Node_1 plays M. Node_1 collects the lists [3, 15] and [14] from Node_2 and Node_3, and then merges them to obtain the final result [3, 14, 15].

Note that the code for parallel merge sort closely resembles the structure of a standard sequential merge sort. However, thanks to role annotations, we obtain a parallel implementation. We will come back to this aspect in Section 6.1.1. Another key benefit is that the compiled code is deadlock-free by construction, as usual for choreographic programming [Carbone and Montesi 2013]. A more detailed discussion of safety and liveness properties is given in Section 7.

4 IMPLEMENTATION

We discuss the main elements of the implementation of Choral. First, we show its syntax and comment on the main differences with Java's. Then, we present the Choral type checker, including examples of the main errors related to roles that it detects and related error messages. Finally, we describe the key components of the Choral compiler.

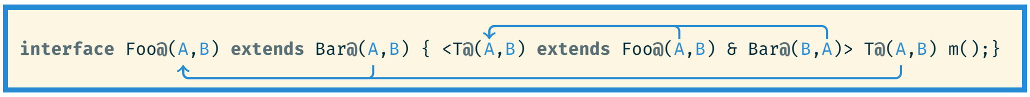
4.1 Language

Figure 5 displays the grammar of Choral; dashed underlines denote optional terms and solid overlines denote sequences of terms of the same sort. We omit syntax for packages and imports, which is as in Java. Reserved identifiers like `super` and `this` are considered identifiers in the grammar, but our compiler treats them like their Java counterparts. The key syntactic novelties are underlined; they consist of (i) syntax for declaring and instantiating role parameters and (ii) the forward chaining operator `>>` (cf. Section 2).

Role parameters have a separate namespace and always appear in expressions like `@(`$\mathtt{A}_1$`, ..., `$\mathtt{A}_n$`)` that follow the name of a class, interface, enum, or type parameter e.g., `DiChannel@(A,B)`. Moreover, role parameters are introduced only by the declaration of a type (e.g., `class Foo@(A,B)`) or a type parameter (e.g., `<T@(A,B) extends Foo@(A,B) & Bar@(B,A)>`) and their scope is limited to the defining type, similar to type parameters in Java. The snippet below contains an example of shadowing of role parameters; for each use of role A, we show its binding site with an arrow.

```
Literals          lit        ::=  null@(A) | true@A | false@A | "a"@A | ... | 1@A | ...
Program           P          ::=  P Interface | P Class | P Enum | P EOF
Enum              Enum       ::=  AN MD enum id@A{id}
Interface         Interface  ::=  AN MD interface id@(A)⟨FTP⟩ extends TE, TE{MDef;}
Annotation        AN         ::=  @id(id = lit)
Modifiers         MD         ::=  public | protected | private | abstract | final | static
Formal
Type Param.       FTP        ::=  id@(A) extends TE & TE
Type Expr.        TE         ::=  id⟨TE⟩ | id@(A)⟨TE⟩ | void
Method Def.       MDef       ::=  AN MD ⟨FTP⟩ TE id (TE id)
Class             Class      ::=  AN MD class id@(A)⟨FTP⟩ extends TE implements TE, TE
                                  {CField CConst MDef; MDef{Stm}}
Class Field.      CField     ::=  AN MD TE id;
Class Con.        CConst     ::=  AN MD ⟨FTP⟩ id(TE id){Stm}
Statement         Stm        ::=  nil | return Exp; | Exp;Stm | TE id = Exp;Stm
                              |   Exp AsgOp Exp; Stm | if(Exp){Stm}else{Stm} Stm
                              |   {Stm} Stm | try{Stm}catch(TE id){Stm} Stm
Expression        Exp        ::=  lit | FAcc | Exp BinOp Exp | Exp.Exp | ⟨TE⟩id(Exp)
                              |   new ⟨TE⟩id@(A)⟨TE⟩(Exp) | id@(A).⟨TE⟩id(Exp) | Exp >> EChain
Field Acc.        FAcc       ::=  id | id@(A).id
Exp. Chain        EChain     ::=  FAcc.id::id | id@(A)⟨TE⟩::new
Assign Op.        AsgOp      ∈    {=, +=, -=, *=, /=, &!=, |!=, %!=}
Binary Op.        BinOp      ∈    {||, &&, |, &, ==, !=, <, >, <=, >=, +, -, *, /, %}
```

Fig. 5. Syntax of the Choral language.



```
interface Foo@(A,B) extends Bar@(A,B) { <T@(A,B) extends Foo@(A,B) & Bar@(B,A)> T@(A,B) m();}
```

The Choral type checker covers all common Java type errors (illegal type conversions, access to type members, etc.), as exemplified below. Indeed, when checking a Choral program with exactly one role parameter, the Choral type checker acts exactly like its Java counterpart.

```
Integer@A x = "foo"@A;
------------^
Incompatible types: expecting 'Integer@A' found 'String@A'.

return x.length();
---------^
Cannot resolve method 'length' in 'Integer@(A)'
```

Roles. The novelties compared to Java compilers emerge when two or more roles are involved. In these settings, programmers can make new kinds of errors that are specifically about the misuse of role-parameterised types—therefore, these errors are pertinent to Choral. In many of the examples discussed so far, we can think of role parameters as Java generics. Although this is a reasonable approximation, some care is necessary when handling type instantiation due to some substantial differences between role and type parameters.

One type of these errors is that data types are instantiated using incompatible roles. Instances of the same type with different role parameters represent values located at different roles, which restricts their usage.

```
String@A x = "foo"@B; // error, same local type but at different roles
-----------^
Incompatible types: expecting 'String@A' found 'String@B'.
```

The order in which roles appear carries meaning, since role parameters are positional—like type parameters in Java generics.

```
void m(SymChannel@(A,B)<T> x) {
  SymChannel@(A,B)<T> a = x; // matching roles
  SymChannel@(B,A)<T> b = x; // error, same roles but wrong positions
------------------------^
Incompatible types: expecting 'SymChannel@(B,A)<T>' found 'SymChannel@(A,B)<T>'.
```

Differently from Java generics, role parameters cannot appear multiple times in the same type, since this corresponds to requiring that the same participant plays multiple roles in the same choreography. In the snippet below, A must play both the sender and receiver for the directed channel c.

```
DiChannel@(A,A)<String> c;
-------------^
Illegal type instantiation: role 'A' must play exactly one role in 'DiChannel'.
```

Forbidding role aliasing is an established restriction in choreographic programming, since aliasing introduces self-communication, which would potentially break deadlock-freedom and the capability to produce separate code for each role (unless roles are provably not-aliased).

Subtyping. Choral types form a hierarchy defined following the same principles used by Java. This hierarchy is used to check if values are compatible type as expected.

```
void m(BiChannel@(A,B)<T,T> x) {
  DiChannel@(A,B)<T>  a=x; // BiChannel@(A,B)<T,T> extends DiChannel@(A,B)<T>
  DiChannel@(B,A)<T>  b=x; // BiChannel@(A,B)<T,T> extends DiChannel@(B,A)<T>
  SymChannel@(A,B)<T> c=x; // error, BiChannel@(A,B)<T,T> does not extend SymChannel@(A,B)<T>
-----------------------^
Incompatible types: expecting 'SymChannel@(A,B)<T>' found 'BiChannel@(A,B)<T>'.
```

Classes and interfaces define their supertypes by extending and implementing other classes and their interfaces with the same set of roles. This restriction provides a substitution principle that elicits all roles involved in a choreography.

```
interface AuditedDiChannel@(A,B,Auditor)<T@C> extends DiChannel@(A,B)<T> {/*...*/}
------------------------------------------------------^
 Illegal inheritance: 'AuditedDiChannel@(A,B,Auditor)' and 'DiChannel@(A,B)<T>' must have the
      same roles.

interface ReplicatedList@(A,Replica)<T@B> extends List@A<T> {/*...*/}
--------------------------------------------------^
 Illegal inheritance: 'ReplicatedList(A,Replica)' and 'List@A<T>' must have the same roles.
```

In some cases, "hidden roles" in choreographies might be useful, e.g., to add external auditing or data replication as an extension of an existing choreography. Unfortunately, this introduces security concerns (channels may have hidden bystanders) or complex communication semantics (what is the meaning of sending a `ReplicatedList@(A,B)` over a channel expecting a `List@A`?). These are general open problems for choreographies, left to future work.

Cyclic inheritance is not allowed and the type checker does not discriminate over role parameters. As an example, consider the `SymChannel` interface; given its symmetric nature, one might be tempted to force this equality by having `SymChannel@(A,B)` subtype `SymChannel@(B,A)`.

```
interface SymChannel@(A,B)<T@C> extends SymChannel@(B,A)<T> { /* ... */ }
----------------------------------------^
Cyclic inheritance: 'SymChannel' cannot extend 'SymChannel'.
```

However, allowing declarations like the one above in Choral would result in cyclic inheritance errors in Java, as exemplified by the following "manual" compilation of the code above.

```
interface SymChannel_A<T> extends SymChannel_B<T> { /* ... */ } // Projection for A
interface SymChannel_B<T> extends SymChannel_A<T> { /* ... */ } // Projection for B
```

To have channels that are instances of both `SymChannel@(A,B)` and `SymChannel@(B,A)` one needs to define a subtype of both as in the snippet below.

```
interface PeerChannel@(A,B)<T@C> extends SymChannel@(A,B)<T>, SymChannel@(B,A)<T> {
  <S@C extends T@C> S@B com(S@A m);          // inherited
  <S@C extends T@C> S@A com(S@B m);          // inherited
  <S@C extends Enum@C<S>> S@B select(T@A m); // inherited
  <S@C extends Enum@C<S>> S@A select(T@B m); // inherited

  PeerChannel@(B,A)<T> flip();               // roles A and B are interchangeable
}
```
Choral Code

By returning an instance of the same interface but with the roles flipped, the method `flip()` introduced by the interface `PeerChannel@(A,B)`, prescribes that the roles `A` and `B` are interchangeable peers.

Finally, primitive types (`int@A`, `bool@A`, etc.) follow the same rules of Java for subtyping, conversions, autoboxing, and autounboxing (when roles match, otherwise the compiler will return a role mismatch error).

Overloading. The Choral type checker refines overload equivalence: it can discriminate overloaded methods by considering roles. For example, `m(Char@B x)` and `m(Char@A x)` can be distinguished, because one parameter is located at `A` whereas the other at `B`. However, we need to be careful with preventing potential clashes in the compiled Java code. Consider the following snippet and error message.

```
class Foo@(A,B) {
  void m(Char@B x) { /* ... */ }  //        void m() at A and void m(Char x) at B
  void m(Char@A x) { /* ... */ }  //        void m(Char x) at A and void m() at B
  void m(Long@A x) { /* ... */ }  // error, void m(Long x) at A and void m() at B
-------^
Illegal overload: 'm(Long@A x)' and 'm(Char@A x)' have the same signature for role 'B'.
```

The last two signatures are distinguishable in Choral, since each method has different parameter types. However, this information is only available to role A, while the projection of both signatures at role B coincide (they would both be `void m()`). This is an instance of knowledge of choice but, differently from conditionals, it cannot be addressed locally (within the class/interface), because extending classes may introduce new branches and new points of choice by overriding and overloading, as in the example below.

```
class Bar@(A,B) extends Foo@(A,B) { void m(Integer@A x) { /* ... */ } }
```
Choral Code

Exceptions. Like every other type lifted from Java, exceptions are located at one role. This design choice allows us to preserve the expected type hierarchy in the generated code and have `java.lang.Exception` as the supertype of all exceptions. The Choral compiler then enforces that a `try-catch` block is located at exactly one role.

```
String@A fetch(DiChannel@(A,B)<T> ch, String@B file) {
  try { return RemoteReader@(A,B).read(ch,file); } catch (IOException@B e) { return null@A; }
--^
Non-local try-catch: try-catch must be at a single role, found 'A' and 'B'.
```

Allowing multiple roles in the mechanics of exceptions introduces a knowledge-of-choice situation where all roles need to obtain information about which handler to execute, if any, and when. The general problem of exceptions and their choreographic handling has been investigated in some theories, but all models proposed so far assume reliable communications among all roles and either rely on specialised orchestration primitives in the target language (i.e., some form of middleware) [Carbone 2009; Carbone et al. 2008; Fowler et al. 2019] or synthesise new communications for recovery [Neykova and Yoshida 2017].

4.2 Compiler

The Choral compiler consists of several steps organised in a pipeline, which we illustrate in Figure 6. From left to right, the first step is (as expected) parsing the input Choral source code to obtain an Abstract Syntax Tree (AST)—the chain operator `>>` is desugared in this step. Next, we perform type checking as previously discussed. This step also annotates the nodes in the AST with type information, which is used in the following projection step. The next step—projection—transforms this annotated AST into a collection of Choral ASTs, each representing the implementation of a single role. At this stage, all types are located at exactly one role, representing the fact that all code is fully local. Finally, in the last step, we output Java code by erasing all role annotations.

We discuss the most important aspects of projection. For clarity, we model and present it as a partial function from (well-typed) Choral terms to Java code: the projection of a Choral term *Term* on a role A, written $(\!| Term |\!)^{A}$, is a Java term that implements the behaviour of A in *Term*.

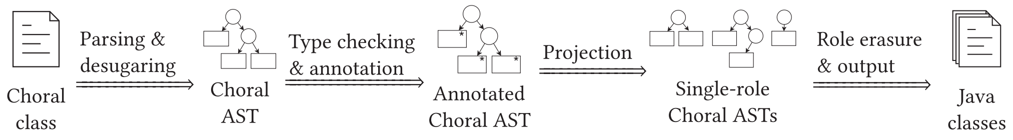


Fig. 6. Pipeline schema of the choral compiler.

Intuitively, this passage covers the last two steps in Figure 6. The full definition of projection is given in Appendix A.

The projection of a Choral `class`, `interface`, or `enum` generates a corresponding Java term for each role parameter. If there are two or more roles, then each Java artefact name is suffixed with the role that it implements, e.g., the Java class compiled from `class Foo(A,B)` for role `A` is called `Foo_A`. If the Choral class has exactly one role, then we use the same name, e.g., `class Integer@A` becomes `class Integer`—this practice minimises the friction of integrating Java types within Choral.

The projection $(\!| TE |\!)^{\mathtt{A}}$ of a type expression *TE* at a role `A` is recursively defined below—we use the auxiliary function roleName(id, i) to retrieve the name of the ith role parameter from the definition of *id*.

$$(\!| id\mathtt{@(}\overline{\mathtt{B}}\mathtt{)<}\overline{TE}\mathtt{>} |\!)^{\mathtt{A}} = \begin{cases} id\mathtt{<}\overline{(\!| TE |\!)^{\mathtt{A}}}\mathtt{>} & \text{if } \overline{\mathtt{B}} = \mathtt{A}, \\ id\_A'\mathtt{<}\overline{(\!| TE |\!)^{\mathtt{A}}}\mathtt{>} & \text{if } \mathtt{A} \text{ is the } i\text{th element of } \overline{\mathtt{B}} \text{ and roleName}(id, i) = A', \\ \mathtt{Unit} & \text{otherwise.} \end{cases}$$

The projection $(\!| Exp |\!)^{\mathtt{A}}$ of an expression *Exp* at role `A` is defined following a similar intuition: it is a recursive stripping of role information as long as `A` occurs in the type of *Exp* or any of its subterms (written $\mathtt{A} \in \text{rolesOf}(Exp)$), otherwise it is the only instance of the singleton `Unit` (stored in its static field `id`), as illustrated by the cases of static field access and constructor invocation below:

$$(\!| id\mathtt{@(}\overline{\mathtt{B}}\mathtt{)}\,.\,f |\!)^{\mathtt{A}} = \begin{cases} (\!| id\mathtt{@(}\overline{\mathtt{B}}\mathtt{)} |\!)^{\mathtt{A}}\,.\,f & \text{if } \mathtt{A} \in \text{rolesOf}(f), \\ \mathtt{Unit.id} & \text{otherwise,} \end{cases}$$

$$(\!| \mathtt{new}\cdot\langle\overline{TE}\rangle id\mathtt{@(}\overline{\mathtt{B}}\mathtt{)}\langle\overline{TE}\rangle\mathtt{(}\overline{Exp}\mathtt{)} |\!)^{\mathtt{A}} = \begin{cases} \mathtt{new}\cdot\langle\overline{(\!| TE |\!)^{\mathtt{A}}}\rangle(\!| id\mathtt{@(}\overline{\mathtt{B}}\mathtt{)}\langle\overline{TE}\rangle |\!)^{\mathtt{A}}\mathtt{(}\overline{(\!| Exp |\!)^{\mathtt{A}}}\mathtt{)} & \text{if } \mathtt{A} \in \overline{\mathtt{B}}, \\ \mathtt{Unit.id(}\overline{(\!| Exp |\!)^{\mathtt{A}}}\mathtt{)} & \text{otherwise.} \end{cases}$$

The projection $(\!| Stm |\!)^{\mathtt{A}}$ of a statement *Stm* at `A` is defined following the above intuition, save for the cases of conditionals and selections, which require care to address knowledge of choice (cf. Section 2.3). Specifically, the rule for projecting `if` statements: for the role evaluating the guard (read from its type), it preserves the conditional; for all other roles, the `if` disappears and it is replaced by the projection of the guard (since it might have side effects) followed by the *merging* $\sqcup$ of the projections of the bodies of the two branches and the projection of the continuation *Stm*.

$$(\!| \mathtt{if(}Exp\mathtt{)\{}Stm_1\mathtt{\}else\{}Stm_2\mathtt{\}}Stm |\!)^{\mathtt{A}}$$
$$= \begin{cases} \mathtt{if(}(\!| Exp |\!)^{\mathtt{A}}\mathtt{)\{}(\!| Stm_1 |\!)^{\mathtt{A}}\mathtt{\}else\{}(\!| Stm_2 |\!)^{\mathtt{A}}\mathtt{\}}(\!| Stm |\!)^{\mathtt{A}} & \text{if } Exp\colon \mathtt{boolean@A}, \\ (\!| Exp |\!)^{\mathtt{A}};\left\{(\!| Stm_1 |\!)^{\mathtt{A}} \sqcup (\!| Stm_2 |\!)^{\mathtt{A}}\right\}(\!| Stm |\!)^{\mathtt{A}} & \text{otherwise.} \end{cases}$$

The merge operator $Stm \sqcup Stm'$ is a partial operator that tries to combine branching code [Carbone et al. 2012], which we adapt to Java for the first time. Essentially, given two Java terms, merging recursively requires them to be equivalent *unless* they are switch statements. Appendix A contains the full definition of merging. Here, we report its most interesting case, merging switch statements:

$$\begin{array}{l} \texttt{switch}\ (\mathit{Exp})\{ \\ \quad \texttt{case}\ id_a\ \texttt{->}\{\mathit{Stm}_a\} \\ \quad \dots \\ \quad \texttt{case}\ id_x\ \texttt{->}\{\mathit{Stm}_x\} \\ \quad \overline{\texttt{case}\ id_y\ \texttt{->}\{\mathit{Stm}_y\}} \\ \quad \texttt{default}\ \texttt{->}\{\mathit{Stm}_{d1}\} \\ \}\ \mathit{Stm} \end{array} \sqcup \begin{array}{l} \texttt{switch}\ (\mathit{Exp})\{ \\ \quad \texttt{case}\ id_a\ \texttt{->}\{\mathit{Stm}'_a\} \\ \quad \dots \\ \quad \texttt{case}\ id_x\ \texttt{->}\{\mathit{Stm}'_x\} \\ \quad \overline{\texttt{case}\ id_z\ \texttt{->}\{\mathit{Stm}_z\}} \\ \quad \texttt{default}\ \texttt{->}\{\mathit{Stm}_{d2}\} \\ \}\ \mathit{Stm}' \end{array} = \begin{array}{l} \texttt{switch}\ (\mathit{Exp} \sqcup \mathit{Exp}')\{ \\ \quad \texttt{case}\ id_a\ \texttt{->}\{\mathit{Stm}_a \sqcup \mathit{Stm}'_a\} \\ \quad \dots \\ \quad \texttt{case}\ id_x\ \texttt{->}\{\mathit{Stm}_x \sqcup \mathit{Stm}'_x\} \\ \quad \overline{\texttt{case}\ id_y\ \texttt{->}\{\mathit{Stm}_y\}} \\ \quad \overline{\texttt{case}\ id_z\ \texttt{->}\{\mathit{Stm}_z\}} \\ \quad \texttt{default}\ \texttt{->}\{\mathit{Stm}_{d1} \sqcup \mathit{Stm}_{d2}\} \\ \}\ \mathit{Stm} \sqcup \mathit{Stm}' \end{array}$$

Above, the merging of two switch statements is a switch whose guard is the merging of the original guards ($\mathit{Exp} \sqcup \mathit{Exp}'$). In the merging, for each case present in both the input switches ($id_a, \dots, id_x$), we get a case in the result whose body merges the respective bodies of the original cases; all cases that are not shared are simply put in the result as they are (the lists of cases $\overline{\texttt{case}\ id_y\ \texttt{->}\ \mathit{Stm}_y}$ from the first and $\overline{\texttt{case}\ id_z : \mathit{Stm}_z}$ from the second).

An example of the result of merging was presented for `DistAuth_Client` in Section 3.1, where the cases for `OK` and `KO` are combined from the respective projections for `Client` of the two branches in the source choreographic conditional evaluated by `IP`. These cases are produced by the rule for projecting selections, which applies to statements of the form $\mathit{Exp}\texttt{;}\ \mathit{Stm}$ when Exp calls (possibly in a chain call) a method annotated with `@SelectionMethod`. (Our type checker checks that these annotations are used only for methods that take enumerated types as parameters, cf. Section 2.3.) For compactness, let $S = \overline{\mathit{Exp}.}\langle\overline{\mathit{TE}}\rangle id_1(id_2\texttt{@A}'.id_3)$ where `@SelectionMethod` $\in$ $\texttt{annotations}(id_1)$

$$\begin{array}{l} (\!| S\,;\mathit{Stm} |\!)^{\mathtt{A}} \\ = \begin{cases} \texttt{switch}((\!| S |\!)^{\mathtt{A}}) \left\{ \begin{array}{l} \texttt{case}\ id_3\,\texttt{->}\{(\!| \mathit{Stm} |\!)^{\mathtt{A}}\} \\ \texttt{default}\ \texttt{->}\{\texttt{throw new ...}\} \end{array} \right\} & \text{if } S\text{: }\texttt{Enum<T>@A} \text{ for some } \texttt{T}, \\ (\!| S |\!)^{\mathtt{A}}; (\!| \mathit{Stm} |\!)^{\mathtt{A}} & \text{otherwise.} \end{cases} \end{array}$$

For the recipient of the selection (first case), the statement becomes a switch on the projection of the *Exp*ression that will receive the selection, while the projection of the continuation *Stm* becomes the body of the corresponding case in the argument. The projection for the other roles (second case) is standard, projecting the *Exp*ression followed by the projection of the continuation *Stm*.

Our implementation of merging is smart enough to deal with some "non-effectful" usages of `Unit`. For instance, consider the following choreography:

```
if(true@A){System@A.out.println("true"@A);}.
```

If we project it at a role different from `A`, say `B`, then we obtain the code `Unit.id(Unit.id)` for the then-branch, and [*blank*] for the (missing) else-branch. These fragments are not mergeable, but our compiler uses a *unit-normalising operator*, given in Appendix A, which transforms also the first fragment into [*blank*] by removing the irrelevant usages of `Unit`.

5 TESTING

Testing implementations of choreographies is challenging, since the distributed programs of all participants need to be integrated (integration testing). We introduce ChoralUnit, a testing tool that enables the writing of integration tests as simple unit tests for choreographic classes.

Writing tests. Following standard practice in object-oriented languages and inspired by JUnit, tests in ChoralUnit are defined as methods marked with a `@Test` annotation [Hamill

2004; Murphy VII et al. 2004]. For example, we can define the following unit test for the `VitalsStreaming` class from Section 3.2.

```
 1 public class VitalsStreamingTest@(Device,Gatherer) {
 2  @Test
 3  public static void test1(){
 4   SymChannel@(Device,Gatherer)<Object> ch =
 5    TestUtils@(Device,Gatherer).newLocalChannel("VST_channel1"@[Device,Gatherer]);
 6   new VitalsStreaming@(Device,Gatherer)(ch, new FakeSensor@Device())
 7    .gather(new PseudoChecker@Gatherer());
 8  }
 9 }
10
11 class PseudoChecker@R implements Consumer@R<Vitals> {
12  public void accept(Vitals@R vitals){
13   Assert@R.assertTrue("bad pseudonymisation"@R, isPseudonymised(vitals));
14  }
15  private static Boolean@R isPseudonymised(Vitals@R vitals) { /* ... */ }
16 }
17
18 class FakeSensor@R implements Sensor@R { /* ... */ }
```
Choral Code

The test method `test1` checks that data is pseudonymised correctly by `VitalsStreaming` . Test methods must be annotated with `@Test`, be **`static`**, have no parameters, and return no values.

At lines 4 and 5, we create a channel between the `Device` and the `Gatherer` by invoking the `TestUtils.newLocalChannel` method, which is provided by ChoralUnit as a library to simplify the creation of channels for testing purposes. This method returns an in-memory channel, which both `Device` and `Gatherer` will find by looking it up in a shared map under the key `"VST_channel1"`. Thus, both roles must have the same key in their compiled code, which is guaranteed, here, by the fact that the expression `"VST_channel1"@[Device,Gatherer]` is syntactic sugar for `"VST_channel1"@Device, "VST_channel1"@Gatherer`.

At lines 6 and 7, we create an instance of `VitalsStreaming` (the choreography we want to test). We use a `FakeSensor` object to simulate a sensor that sends some data containing sensitive information (omitted). We then invoke the `gather` method, passing an implementation of a consumer that checks whether the data received by the `Gatherer` has been pseudonymised correctly.

Given a class like `VitalsStreamingTest`, ChoralUnit compiles it by invoking our compiler with a special flag (`-annotate`). This flag makes the compiler annotate each generated Java class with an `@Choreography` annotation that contains the name of its source Choral class and the role that the Java class implements.

When the compilation is finished, we can invoke ChoralUnit to run the tests from the class `VitalsStreamingTest`. Once launched, the tool finds all the Java classes with an `@Choreography` annotation whose `name` value corresponds to `VitalsStreamingTest`. By construction, each discovered class has a method with the same name, corresponding to the namesake method from the source Choral test class (`test1`, in our example). ChoralUnit exhaustively runs all the tests found in the test class. Namely, for each `@Test`-annotated method and for each class generated from the Choral source, ChoralUnit starts a thread that runs the local implementation of that method implemented by that class.

In our example, `VitalsStreamingTest` is compiled to a class for `Device` and another for `Gatherer`, each with a `test1` method. Thus, ChoralUnit starts two threads, one running `test1` of the first generated Java class and the other running `test1` of the second generated Java class.

Multiparty assertions. In the previous example, we have written an assertion (in `PseudoChecker`) that checks a condition at a single role (`Gatherer`). Sometimes, it is useful to assert conditions that involve multiple roles. A typical example is testing the correct implementation of protocols that aim at making two parties agree on a symmetric cryptographic key, like the Diffie-Hellman protocol [Diffie and Hellman 1976]. In particular, after running the protocol, the two participants (say `A` and `B`) should have the same key. We can express this assertion as follows.

```
Assert2@(A,B).assertEquals("key mismatch"@B, chAB, keyA@A, keyB@B);
```
Choral Code

Above, method `assertEquals` of class `Assert2` uses the channel `chAB` to communicate the key at `A` (`keyA`) from `A` to `B`, and then checks locally at `B` that it is equal to the key at `B` (`keyB`). If the check fails, then an assertion error is raised at `B`.

Class `Assert2` can be user-defined, and likewise, developers can define classes that allow for assertions that involve more roles (e.g., `Assert3`, `Assert4`, etc.). In these implementations, the user can also freely code different protocols for communicating the data among the participants.

6 EVALUATION

In Section 3, we explored how Choral can be used to program choreographies for a few realistic scenarios. In this section, we extend the evaluation of our approach in three different directions:

(1) In Section 6.1, we exemplify how one can use Choral to transition existing (Java) programs to choreographies. In Section 6.1.1, we show how Choral aids in transitioning sequential algorithms into concurrent implementations. We consider a Java algorithm and present the necessary steps to transform it into a Choral program that distributes its computation over three nodes—the number of nodes follows naturally from the recursive structure of the algorithm. The steps are straightforward, thanks also to the guidance offered by the Choral compiler. In Section 6.1.2, we consider a complete, three-tier system: RetwisJ,[2] a clone of Twitter implemented by the Spring team as an example of integration with the Redis data store. RetwisJ comes as a monolithic application that consists of mainly three components that, respectively, handle clients' invocations, the business logic, and the interaction with a data store. Following this design, in the transition to Choral (called ChoRetwis), each component appears as a role in the choreography. ChoRetwis is a drop-in replacement for RetwisJ (e.g., we can use RetwisJ and ChoRetwis with the same clients and data store). As an advantage of our choreographic refactoring, the architecture is more flexible wrt deployment: all components can be deployed on the same or different machines.

(2) In Section 6.2, we compare Choral to a popular alternative for concurrent and distributed programming: reactive actors. We use the Akka framework for Java as a representative for reactive actors. Since Choral is essentially an extension of Java, this choice helps in comparing our approach against more standard approaches to concurrent programming at the net of linguistic differences. In addition to the key qualitative advantage that Choral provides choreography compliance, we find that Choral contributes to keeping the codebase smaller. Furthermore, we find that the Java code generated by the Choral compiler is not significantly

[2]https://github.com/spring-attic/spring-data-keyvalue-examples/tree/master/retwisj

```java
public static Long multiply(Long n1, Long n2) {
  if (n1 < 10 || n2 < 10) {
    return n1 * n2;
  } else {
    Double m = Math.max(Math.log10(n1), Math.log10(n2)) + 1;
    Integer m2 = Double.valueOf(m / 2).intValue();
    Integer splitter = Double.valueOf(Math.pow(10, m2)).intValue();
    Long h1 = n1 / splitter; Long l1 = n1 % splitter;
    Long h2 = n2 / splitter; Long l2 = n2 % splitter;
    Long z0 = Karatsuba.multiply(l1, l2);
    Long z2 = Karatsuba.multiply(h1, h2);
    Long z1 = Karatsuba.multiply(l1 + h1, l2 + h2) - z2 - z0;
    return z2 * splitter * splitter + z1 * splitter + z0;
  }
}
```

```
public static Long@A multiply (Long@A n1, Long@A n2,
  SymChannel@(A, B)<Long> ch_AB,
  SymChannel@(B, C)<Long> ch_BC,
  SymChannel@(C, A)<Long> ch_CA) {
  if (n1 < 10@A || n2 < 10@A) {
    ch_AB.<Choice>select(Choice@A.DONE); ch_CA.<Choice>select(Choice@A.DONE);
    return n1 * n2;
  } else {
    ch_AB.<Choice>select(Choice@A.REC); ch_CA.<Choice>select(Choice@A.REC);
    Double@A m = Math@A.max(Math@A.log10(n1), Math@A.log10(n2)) + 1@A;
    Integer@A m2 = Double@A.valueOf(m / 2@A).intValue();
    Integer@A splitter = Double@A.valueOf(Math@A.pow(10@A, m2)).intValue();
    Long@A h1 = n1 / splitter; Long@A l1 = n1 % splitter;
    Long@A h2 = n2 / splitter; Long@A l2 = n2 % splitter;
    Long@A z0 = Karatsuba@(B, C, A)
      .multiply(ch_AB.<Long>com(l1), ch_AB.<Long>com(l2), ch_BC, ch_CA, ch_AB)
      >> ch_AB::<Long>com;
    Long@A z2 = Karatsuba@(C, A, B)
      .multiply(ch_CA.<Long>com(h1), ch_CA.<Long>com(h2), ch_CA, ch_AB, ch_BC)
      >> ch_CA::<Long>com;
    Long@A z1 = Karatsuba@(A, B, C)
      .multiply(l1 + h1, l2 + h2, ch_AB, ch_BC, ch_CA) - z2 - z0;
    return z2 * splitter * splitter + z1 * splitter + z0;
  }
}
```

Fig. 7. Karatsuba algorithm. Left: Java (sequential). Right: Choral (choreographic).

different in size from manually written Java code. This opens the door to a quantitative evaluation, which we carry out in Section 6.3.

(3) In Section 6.3, we present a quantitative evaluation of how Choral impacts software development and execution performance. In Section 6.3.1, we report relevant measurements on the performance of the Choral compiler. Our analysis shows that using Choral leads to smaller codebases and that the numbers of roles and conditionals in a choreography are significant determinants of how much our approach contributes to this aspect. We also observe that both our type checker and our compiler are fast when used on our set of examples—they provide feedback and complete the compilation in a matter of milliseconds. Thus, our approach does not significantly reduce the performance of development toolchains. In Section 6.3.2, we look at the runtime performance of the code that we generate. Specifically, we compare the execution times of Choral and Akka implementations of the Karatsuba algorithm presented, respectively, in Sections 6.1 and 6.2. We find the performance of the two models comparable (and the Choral variant performs the best in the majority of the tested cases).

Overall, our results are encouraging. In addition to the advantage of choreography compliance, smaller codebases tend to host fewer bugs [Bessey et al. 2010], and Choral appears rather approachable when we consider the context of existing practices.

The code used in this evaluation is available in two public repositories. The first contains the material for the comparison with Java and Akka (both quantitative and qualitative) and for benchmarking the performance of the compiler.[3] The second includes the original code and the Choral re-implementation of RetwisJ.[4]

6.1 From Java to Choral

Thanks to the fact that Choral is based on mainstream abstractions, we can use the implementation of a sequential algorithm in Java as a starting point to obtain a concurrent variant (in Choral).

6.1.1 Transforming an Algorithm: Karatsuba. Consider the algorithm for fast multiplication by Karatsuba and Ofman [1962]. We report an implementation of that algorithm in Java on the left side of Figure 7.

[3]https://github.com/choral-lang/evaluation
[4]https://github.com/choral-lang/choretwis

Starting from the Java implementation, we can obtain a distributed implementation of the same algorithm in Choral by adding: (a) information on where the data is located and (b) data transmissions for moving the data and implementing knowledge of choice.

We report the resulting Choral program on the right side of Figure 7, highlighting the additions from the original Java code in yellow.

The Choral program has three roles (`A`, `B`, and `C`), which distribute among themselves the three sub-calculations of the algorithm. In the parameters and return type, we added information on data locality (e.g., `Long@A n1`) and the necessary channels (e.g., `ch_AB`) for moving data in the implementation of the method. Given the original Java code, the type checker of the Choral compiler would assist the programmer by pointing out that data locality information must be added. Likewise, in the implementation of the method, we added information on data locality for constant values and variables (e.g., `Double@A m`). Additionally, we added the necessary data transmissions: selections to implement knowledge of choice for the conditional, and communications of values whenever they should move from a role to another. Again, the Choral compiler aids the programmer by asking for all this information.

6.1.2 Transforming an Entire System: RetwisJ (Redis-based Twitter Clone). We now focus on the transformation of a complex, real-world system to provide a more comprehensive view of the process. The transformation allows us to illustrate how Choral can help developers transition from monolithic to distributed implementations (à la microservices) while maintaining their options open wrt code reuse, interoperability, and deployment configurations.

Concretely, we took RetwisJ,[5] which is a Java, Spring-based port of Retwis,[6] and re-implemented its logic as a distributed application that consists of three separate modules. We report, at the top of Figure 8, the simplified class diagram of RetwisJ. The application is a monolith, where the central module “RetwisController” works both as the gateway for serving webpages to the user (the “JSP pages” module) and as the entry-point for user requests (e.g., to log in, to post tweets, etc.). The classes “User” and “Post” model the main entities in the system, while the “RetwisRepository” implements the logic for data persistence and retrieval.

Refactoring RetwisJ in Choral naturally follows well-known patterns from microservice architectures [Dragoni et al. 2017; Newman 2021]: interaction with the client is handled by a “Gateway” component; business logic is managed by a “Controller” component; and data storage and access is managed by a “Storage” component. These components (depicted in the lower part of Figure 8) correspond to roles in our implementation, so we obtain a choreography that defines how these three roles collaborate to implement the application.

Each component is implemented by combining its respective code compiled from the choreography (for coordination) together with local code that implements the internal functionalities that are out of the scope of the choreography. For such internal functionalities—e.g., the concrete data read/write operations on Redis—we reuse existing code from RetwisJ. In Figure 8, we display which classes from RetwisJ have been reused as-is (e.g., “User” and “Post”). All three components are loosely coupled, in the sense that they interact purely via message passing (as instructed by the choreography). Since the choreography uses our abstract channel interfaces, our implementation is more flexible than the original RetwisJ. Developers can choose to distribute the components by using, e.g., TCP/IP channels, or to deploy all of them as a single application using in-memory communication channels (which is the only option for RetwisJ).

[5] Source code available at https://github.com/spring-projects/spring-data-keyvalue-examples, documentation available at https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/

[6] A Twitter clone originally proposed by the Redis team to illustrate the capabilities of the data store, described at https://redis.io/topics/twitter-clone

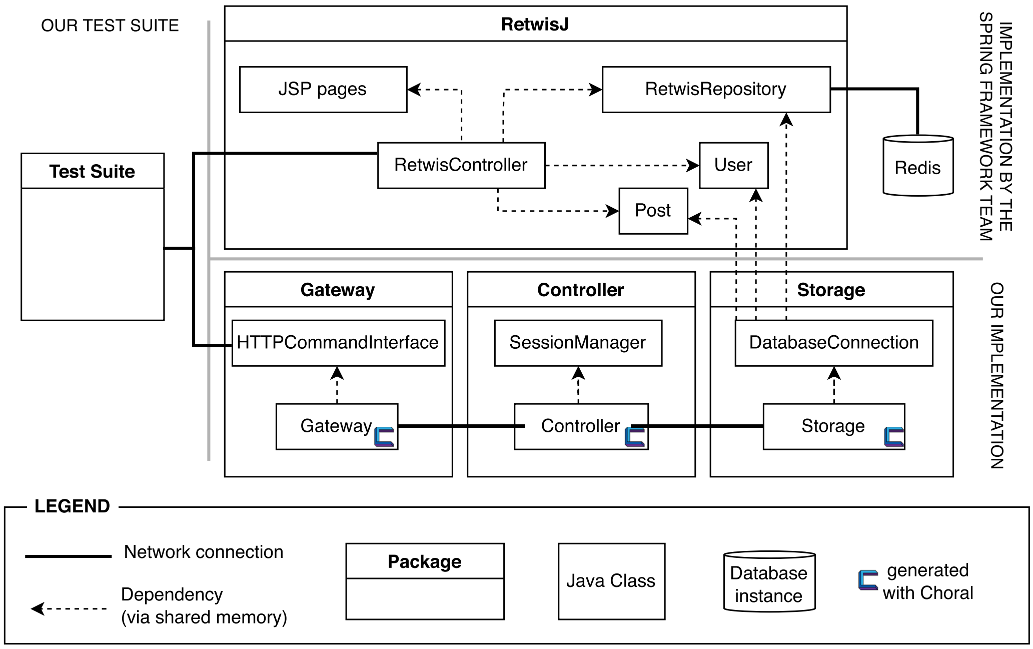


Fig. 8. Diagram of the RetwisJ and ChorRetwis Systems (classes, packages, and deployment).

The choreography is strategically parametric on a few notable aspects.

— The gateway receives API calls through a generic "CommandInterface," which allows us to expose the API over different media. In our concrete example, we implemented an "HTTP-CommandInterface" for exposing a typical REST API (designed by us) and an alternative that acts as a drop-in replacement for RetwisJ by implementing the API expected by the JSP pages provided in that project.
— The controller delegates the storage and retrieval of the session state to an abstract "Session-Manager." Our implementation stores state locally (in memory), but it can in principle be generalised to storing state on an external distributed store, to allow for replication.
— The storage component relies on an abstract "DatabaseConnection" (a database abstraction layer), which determines how data is concretely represented, read, and written. Our implementation reuses the Redis-based code from RetwisJ. Thus, RetwisJ and our implementation can even be used in parallel. Storage is, however, not limited to using Redis and alternative implementations of "DatabaseConnection" can be provided.

To test our implementation and its consistency with the original RetwisJ, we developed a test suite that programmatically invokes the HTTP APIs of the two systems. The suite performs a series of tests that simulate usage, modifying state (e.g., creating users and posts) and then checking that the results are as expected.

6.2 Programming Paradigms: Choral and Akka

We carry out a brief comparison between Choral and an established framework for concurrent programming: the Akka framework for the Java language. Akka is a popular reactive framework based on actors for the "traditional" way of programming concurrent software; that is, software

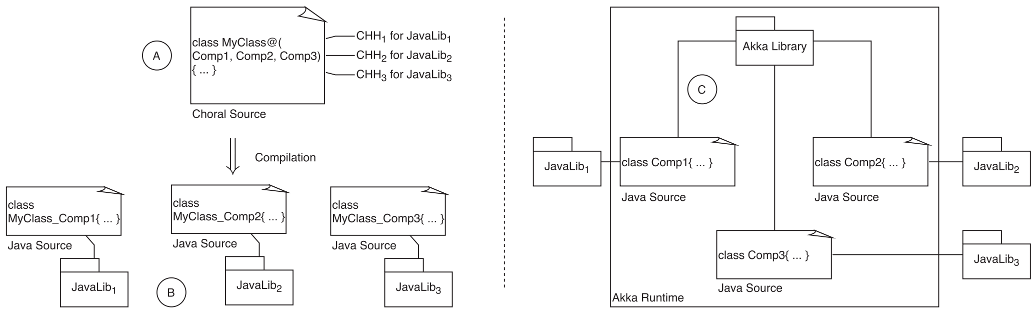


Fig. 9. Depiction of the programming approaches of Choral (left) and Akka (right).

where each endpoint is programmed from a local viewpoint, in contrast with the global view on the expected interactions of choreographic programming. We use Akka version 2.6.18.

We depict the development processes of Choral and Akka in Figure 9, respectively, on the left and the right sides. The processes are slightly different.

— In Choral, we implement a choreography using a single codebase (Ⓐ). The codebase for each participant is then generated automatically by our compiler (Ⓑ).
— By contrast, in Akka, we implement the behaviour of each participant separately (Ⓒ). There are no components to write choreographies.

Choral provides choreography compliance through its language and compiler. Akka provides no tools to express choreographies, nor to check for compliance—these aspects must be handled manually by the programmer.

Both Choral and Akka require the programmer to adopt a few principles, respectively: for Choral, our notion of data types with multiple roles; for Akka, the design patterns and APIs expected by the Akka framework for user implementations, and the APIs of the Akka libraries. Notably, Choral does not fix any APIs. The choice of APIs and implementations of channels or other methods are completely up to the user. Thus, Choral leaves more freedom to the developer in choosing what libraries to rely on. Furthermore, Choral requires no runtime library during execution, while Akka requires programmers to adopt the Akka runtime (represented in Figure 9 by the Akka Runtime rectangle that surrounds the components).

Choral and Akka allow for reusing existing Java code and libraries, e.g., database drivers. When Java code involves a single role, using it in Choral is straightforward—we interpret any Java type as a type parametric at a single role. In general, programmers can explicitly coerce Java types to arbitrary Choral types by using special header files called Choral Headers, shortened as CHH ($\mathrm{CHH}_1, \ldots, \mathrm{CHH}_3$ in Figure 9), which isolate code that might be "unsafe," because it is manually written in Java.

To get more concrete observations and data, we manually implemented in Akka three choreographies presented in this article, namely, DistAuth (see Section 3.1), MergeSort (see Section 3.3), and Karatsuba (see the beginning in this section). In Table 1, we compare the sizes of the three codebases for each example in terms of lines of code (LOC). Specifically, we report: the size of the Choral implementation, the size of the Java code generated from the Choral implementation, and the size of the manually written Akka implementation.

The comparison between the implementations of DistAuth is straightforward. The main difference lies in the fact that Akka follows a reactive programming style that dictates the usage of fields and messages of different types inside each actor to track the (local) status of the protocol. The flow of interactions becomes thus implicit, and needs to be reconstructed by the expected

Table 1. Comparison of Three Codebases Implemented in Choral and Akka

Program	Choral (LOC)	Java/Choral-generated (LOC)	Java/Akka (LOC)
DistAuth	56	137	234
MergeSort	63	239	166
Karatsuba	31	92	118

asynchronous activations of methods at actors—a similar observation has already been made by Weisenburger et al. [2018]. Differently, in Choral the flow of interactions is made explicit by our type system and sequencing of actions. This difference makes the Choral codebase quite shorter, since the fields and message types used to implement causal dependencies in Akka add boilerplate code. The structures of the components between the generated Choral code and the Akka implementation follow the same pattern, i.e., we have one class for each participant (the Client, the Service, and the IP from Section 3.1).

We then moved to implementing the MergeSort and Karatsuba examples, because their recursive nature makes them a good fit for reactive programming as in Akka. Indeed, differently from the DistAuth implementation that had a one-to-one correspondence between the Choral-generated and Akka-based classes, the Akka implementations of MergeSort and Karatsuba rely mainly on a single class that defines the implementation of all roles. We find that the actor paradigm and the programming style we used for Choral in this article promote different ways of dealing with recursion. For the Akka implementation, we followed the idiomatic approach of creating a new actor for each new recursive call of the distributed algorithm. By contrast, in the Choral implementations, it is natural to use the same participants in recursive calls by switching their roles as shown in Sections 3.3 and 6.1.1—this lowers complexity and performance costs wrt coordination, since we avoid creating and managing new participants. Adopting the "one class" style for Akka helps keep the codebase small (the Java implementation generated from Choral is bigger for MergeSort), but makes the implementation tightly coupled.

Interestingly, for the Karatsuba algorithm, the Choral-generated Java implementation is smaller than the manually written Akka implementation. This is because Karatsuba requires more coordination and local tracking of the distributed state, correspondent to boilerplate code that defines bookkeeping message types and fields in Akka.

What went wrong. There are valuable lessons that we learned from the exercise of writing implementations of the Karatsuba algorithm in Akka and Choral. In particular, writing the Akka implementation was trickier and more error-prone. We illustrate two representative issues, which arise from the lack of a choreographic view in Akka. These issues have been analysed by two of the authors working together to peer-review both the development process and the resulting code.

— For the first Akka implementation, we noticed that its performance was always the same for all tested inputs. It took us some time to notice that no communications were performed at all: all multiplications were resolved locally, because there was a typo in the direction of the inequality check that decides whether to perform the `z0-z1-z2` decomposition of the product (cf. Figure 7) or perform it directly. This was a very subtle bug, because the program was terminating successfully and returned the right results.
This issue was caused by the fact that the code for the Karatsuba algorithm needs to be distributed across different methods in Akka, which obfuscates the original structure of the algorithm and adds opportunities for banal bugs (e.g., swapping two arguments by mistake). We did not encounter this kind of issues with the Choral implementation, because we just needed to augment the original (correct) Java implementation with channel usage and roles

Table 2. Performance Results for the Choral Compiler

Program	Choral (LOC)	# Roles	# Conditionals	Java (LOC)	Size Increase (%)	Type Checking (ms)	Proj. Checking (ms)	Projection (ms)
HelloRoles	9	2	0	14	55%	5.915	0.334	0.187
ConsumeItems	16	2	1	49	206%	9.572	0.861	0.607
BuyerSellerShipper	40	3	2	126	215%	8.204	1.274	1.015
DistAuth	56	3	1	137	144%	11.463	9.097	0.986
VitalsStreaming	47	2	1	78	65%	7.864	1.384	0.417
DiffieHellman	26	2	0	36	38%	5.911	0.232	0.152
MergeSort	63	3	4	239	279%	8.517	7.891	3.723
QuickSort	74	3	3	200	170%	7.213	6.204	2.806
Karatsuba	31	3	1	92	196%	6.491	2.566	1.078
DistAuth5	66	5	1	226	242%	10.581	5.573	1.036
DistAuth10	91	10	1	438	381%	10.576	5.643	3.011

(cf. Figure 7). The distribution of code in the final Java implementation has been carried out by the Choral compiler without mistakes.

— Another bug that we encountered was due to a typo, too, which made the Akka implementation perform the calculation of `z2` instead of the one of `z1`. However, more interestingly, this error manifested itself as a deadlock: an actor was waiting for the arrival of the three sub-calculations for `z0`, `z1`, `z2`, and instead received only those for `z0` and `z2` (the latter twice). Fixing this kind of bugs is tricky in parallel programming, and indeed our first attempt at a fix introduced another bug: We fixed the deadlock, but we started obtaining wrong results, because the fix caused a swap of the inputs for `z1` and `z2`.
The key reason behind these bugs was that, in Akka, we had to manually write code that reads and writes tags in messages to know if they contain the result of computing `z0`, `z1`, or `z2`. In this code, errors can be written in the code that creates messages at the sender and the code that processes them at the receiver. Implementations can therefore fall out of sync. Choreographic programming (hence Choral) prevents this kind of bugs entirely, because it is not possible to write mismatched communications at the choreographic level.

6.3 Microbenchmarks

We now move to a more systematic and quantitative evaluation of how Choral impacts software development—in addition to the key benefit of choreography compliance. First, we evaluate the performance of the Choral compiler with microbenchmarks on 11 Choral programs. Then, since we implemented the same Karatsuba algorithm both in Java, Choral, and Akka, we provide some preliminary runtime benchmarks by contrasting their performance.

6.3.1 Compilation Benchmarks. Regarding the performance of the Choral compiler, we report our results in Table 2. There, for each program, we report (left to right): the name of the Choral program, lines of code, number of roles, number of conditionals (**`if`** and **`switch`** blocks), lines of code of the compiled Java code (total for all roles), number of milliseconds to perform type checking, number of milliseconds to perform the check for projectability, and number of milliseconds to perform the projection (Section 4.2). All code is well indented and the number of lines just omits empty ones. We collected times on a machine equipped with an Intel Core i5-3570K 3.4 GHz CPU and 12 GB of RAM, running macOS 10.15 and Java 17. The reported times are averages of 1,000 runs each, after a warm-up of 1,000 prior runs.

Table 2 reports data for programs shown in this article, plus four other programs: BuyerSeller-Shipper is inspired by a recurring e-commerce example found in choreography articles [Carbone et al. 2012; Honda et al. 2016]; the Diffie-Hellman protocol for cryptographic key exchange [Diffie and Hellman 1976]; and DistAuth5 and DistAuth10, which are variants of the DistAuth class from Section 3.1, where we, respectively, add 3 and 7 roles, 2 and 7 channels, and 4 and 14 selections for coordination.

Our preliminary data from Section 6.2 points out that Choral programs are significantly smaller than the Java implementations compiled from them. This is good in itself: recall that smaller codebases typically host fewer bugs [Bessey et al. 2010]. In our microbenchmarks, compilation leads to an average increase of 181% in codebase size (going from the 38% for `DiffieHellman` up to 381% for `DistAuth10`): the difference between the sizes of the original Choral program and the generated Java code is a rough approximation of code that the programmer has been spared from writing manually. Furthermore, our microbenchmarks suggest that two main parameters affect this benefit.

— The number of roles involved in the source choreography. This factor is explained by the fact that each statement in Choral involving n roles corresponds to n statements in the generated Java code—one for each role, implementing what the role has to do to follow the choreography. For example, a Choral statement invoking a method to communicate data from `A` and `B` would produce a statement in the code for `A` (for sending) and a statement in the code for `B` (for receiving). There are examples of choreographies with many instructions that involve a single role, like `DiffieHellman`, which results in a smaller expansion.

— The number of conditionals. Conditionals usually require performing selections to handle knowledge of choice (Section 2.3). Then, the (code compiled for the) roles receiving selections has to inspect the type of the received message using a **`switch`** statement, which is automatically added by our compiler and is not present in the original Choral code. For example, MergeSort and QuickSort differ in that the former has four conditionals whereas the latter has three conditionals, and, respectively, reach an expansion of 279% and 170%.

As a final remark, we observe that type and projectability checking and projection do not add any significant delay to the development experience: they, respectively, average ca. 8.391 ms, 3.732 ms, and 1.365 ms. These performance match our programming experience with Choral, where the compiler managed to feel quite responsive in providing quick feedback while coding. Both operations are mostly influenced by the number of conditionals and roles, which agrees with our previous observations.

6.3.2 Runtime Benchmarks. We conclude this section by comparing the execution times of the Choral and Akka implementations of Karatsuba. We show the results in Figure 10. In the figure, we report in each of the six plots the average execution time, in nanoseconds, of a sequence of 1,000 multiplications. Each quadrant regards a specific "tier" of multiplication, i.e., the 10^9 tier corresponds to the multiplication of two factors of the shape, e.g., $i * 10^5$ and $j * 10^4$, which produce a result of that tier's magnitude. All benchmarked implementations use the same inputs: We generated and used six files, each containing 1,000 pairs of random factors. Each file corresponds to a tier. The considered tiers are: 10^9, 10^{11}, 10^{13}, 10^{15}, 10^{17}, and 10^{19}—the latter approximates the maximal values managed by the Java `long` data type, but we make sure to never produce overflows. We performed the benchmarks on the same machine used to benchmark the Choral compiler above, with Java 17 and Akka 2.6.9. For each implementation, we run the benchmark two times in sequence, discarding the data of the first run to warm up the JVM and provide stabler results.

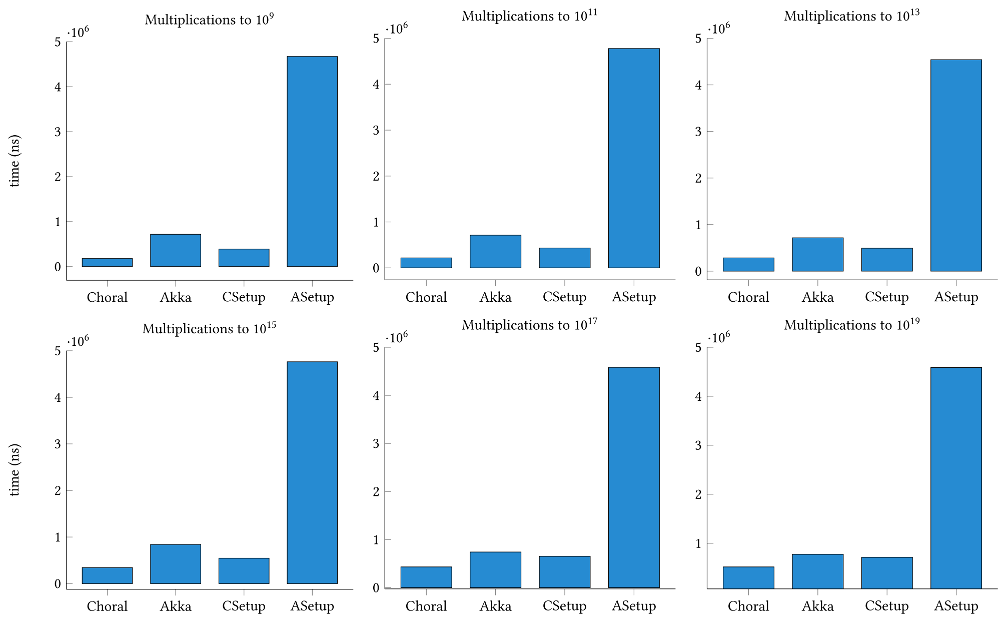


Fig. 10. Benchmarks of Choral (Choral, CSetup), and Akka (Akka, ASetup) implementations of the Karatsuba algorithm.

To benchmark both Akka and Choral implementations, we used in-memory communications. We wrote an implementation of in-memory channels for Choral, while for Akka, we used the default in-memory channels provided by the framework. In Figure 10, we report the execution times of the Choral and Akka implementations both including their setup (respectively, "ASetup" and "CSetup") and without it (respectively, "Akka" and "Choral"). The reason to report setup times is to provide the reader with an indication of the wall-clock times taken by the alternatives. The setup of Akka regards the creation of the `ActorSystem` to execute the Karatsuba behaviour and its closure after having obtained the result. The setup for Choral includes the creation of an `Executor` pool of three threads, the creation of the three in-memory channels, and the closure of the pool after having obtained the result.

The Choral implementation outperforms Akka's in all quadrants, both with and without the overhead from the setup—indeed, although with a slight margin, the performance of the "Choral" implementation *with* the setup times outperforms "Akka" without the corresponding overhead. We also note that, contrarily to Choral, the wall-clock execution times of Akka are largely dominated by its setup times. This phenomenon is testified by the "ASetup" bar in Figure 10, which has almost the same performance irrespective of the magnitude of the computed result, while the alternatives show longer times for greater magnitudes. We attribute this phenomenon to the different ways in which the runtimes of the Choral and Akka in-memory implementations manage concurrency and messaging. Indeed, skimming through the internal code of Akka, we found that the framework puts in place threading models and advanced messaging systems optimised for the execution of many actors that communicate in parallel, which can take a high-performance toll when implementing "lighter" computations, like the one benchmarked here. In the future, we plan to investigate these aspects more in-depth, identifying a set of benchmarks sensible to the peculiarities of the chosen Choral and Akka runtimes and able to help shed light on the trade-offs of either approach.

6.4 Threats to Validity

Our evaluation covers a broad range of topics related to applicability and performance, which we believe indicate that Choral is a promising candidate for the programming of concurrent and distributed systems based on choreographies. Here, we discuss threats to the validity of our evaluation and directions for its future extension.

The experiments that we have presented include an algorithm (Karatsuba), a reference architecture (RetwisJ), one established programming framework (Akka), and several microbenchmarks. These go on top of the other examples and use cases that we discussed in the previous sections, which included other concurrent and distributed scenarios.

Regarding Choral's applicability and usability, we could improve the validity of our evaluation by considering additional kinds of algorithms, architectures, and popular concurrent programming frameworks. Comparisons to other programming frameworks might also benefit from including a wider array of programs. Another interesting direction would be to evaluate Choral's usability by conducting user studies that involve practitioners from academia and industry. These user studies could provide precious input for the future growth and refinement of Choral.

Regarding the performance of Choral's compiler and generated code, a potential threat is that there might be performance bottlenecks that are not covered by our set of benchmarks. An interesting future improvement could be to systematically extend this coverage, by developing a tool that synthesises Choral programs according to different patterns and constraints that are determined by input parameters—such as those that we considered in Table 2. Another potential threat is that there might be parameters that significantly influence performance that we did not consider in Table 2. A tool for synthesising "random" choreographies could be useful in the discovery of such parameters. Possible starting points for the development of this tool could be synthesisers of (i) business processes [Burattin 2016; Burattin and Sperduti 2011] or (ii) choreographies in simplistic languages [Cruz-Filipe et al. 2022].

7 RELATED WORK, DISCUSSION, AND FUTURE WORK

Choral is a *choreographic programming language*, in that it makes the flow of interactions and their related computations manifest from a global viewpoint [Montesi 2013]. While Choral suffices already in tackling different kinds of use cases, as we have discussed in this article, the literature on choreographies is vast. Choral includes and generalises many features found in previous work on choreographies. There are other features that we have not considered in this work, along with open problems that we pointed out when appropriate in the previous sections. We discuss related work and other potential future developments of Choral in the rest of this section.

Previous implementations of choreographic programming. The idea of synthesising local participant specifications that comply with choreographies has been a hot research topic for more than 20 years, and work in this line of research is typically based on automata or process calculi abstractions [Alur et al. 2000; Autili et al. 2018; Basu et al. 2012; Honda et al. 2016; Qiu et al. 2007]. Previous implementations of choreographic programming consist of Chor [Carbone and Montesi 2013] and AIOCJ [Dalla Preda et al. 2017], which are based on process calculi and generate executable Jolie code. Compared to them, Choral solves the modularity problems mentioned in the Introduction, by revisiting choreographies under the light of mainstream abstractions. Another advantage is that the types of channels needed by a choreography are made explicit and can be user-defined [Carbone et al. 2012; Carbone and Montesi 2013; Honda et al. 2016; Qiu et al. 2007].

Other approaches to spatially distributed programming. The types that support our choreography-as-objects interpretation have been inspired by ideas found in modal logics for mobile ambients [Cardelli and Gordon 2000] and, later, in the line of work on multitier programming [Cooper et al.

2006; Liu et al. 2009; Murphy VII et al. 2007, 2004; Neubauer and Thiemann 2005; Serrano et al. 2006; Weisenburger et al. 2018]. In the words of Murphy VII et al. [2004], these works represent other approaches to "spatially distributed computation." For example, in the most recent incarnation of multitier programming (ScalaLoci, by Weisenburger et al. [2018]), a distributed application is essentially defined as a single program that composes different functions, each localised at a single participant. A function can then invoke special primitives to request remote computation by another participant, whose implementation must always be ready for such requests [Weisenburger et al. 2020]. Differently from choreographies, this makes the flow of communications implicit and dependent on an underlying middleware—indeed, multitier programming was not designed to address the problems of defining choreographies and addressing choreography compliance as an aim. Choral generalises data types localised at a single participant to data types localised at many participants (roles), which enables our novel development process for choreography-compliant libraries. Castro-Perez and Yoshida [2020] explored the parallelisation of a simple multitier first-order functional language, for which they can infer abstract (computation is not included) choreographies of the communication flows that these programs can enact; Choral could be a candidate implementation language for this kind of models. Fowler et al. [2019] explored the idea of incorporating simple choreographic languages (without computation) to ensure that multitier code between two participants enacts specific protocols.

Choral's clear relation to the ideas found in logics for mobile ambients has already proven useful. In particular, Giallorenzo et al. [2021] use Choral to kickstart an investigation of the links and differences between choreographic and multitier programming, by taking Choral and ScalaLoci as representative languages. After identifying the core abstractions that differentiate the two approaches, the authors provide algorithms for translating Choral code into ScalaLoci code and vice versa. Going from a multitier program to a choreographic one requires synthesising one of the many possible protocols (a choreography) that implements the necessary communications to execute the multitier program (which does not specify this aspect). This connection paves the way for joint research and cross-fertilisation between the two communities [Giallorenzo et al. 2021].

Higher-order choreographies. Interpreting choreographies as objects enables, for the first time, higher-order composition of choreographies that carry state (the fields of the objects): stateful choreographies (objects) can be passed as arguments. Stateful choreographies have been investigated before without higher-order composition—see, for example, the works by Carbone et al. [2012]; Chen and Honda [2012]; Cruz-Filipe and Montesi [2020]. Demangeon and Honda [2012] studied how parameters that abstract choreographies can be expanded syntactically, but their choreographies cannot carry state and there must be a role that acts as an orchestrator to "enter" into a choreography (whereas in Choral, control is fully distributed). In the setting of multitier programming, Weisenburger and Salvaneschi [2019] introduced a module system to write multitier programs as compositions of submodules. Differently from Choral and the work by Demangeon and Honda [2012], their approach requires fixing roles statically, whereas in our case roles can be freely instantiated at runtime—for example, our merge sort example in Section 3.3 exploits this feature when roles are exchanged in recursive calls. Thus, our new data types might be interesting in the setting of multitier programming too.

Choral's principles in other settings. The core idea in Choral's design is to have data types at multiple locations (the roles of the choreography): `T@(A_1, ... A_n)`. This information is the compass that guided us in lifting the various aspects of Java to the choreographic level and in designing our notion of projection. We believe that this idea can be easily applied to other object-oriented languages—C#, Kotlin, Scala, and others—by extending their types in the same way and retracing our steps.

The applicability of this core idea goes even beyond object-oriented languages. Choral's first technical report and release [Choral Development Team 2020; Giallorenzo et al. 2020] have already inspired the investigation of theories and implementations of functional choreographic programming languages [Cruz-Filipe et al. 2022, 2023a; Graversen et al. 2023; Hirsch and Garg 2022; Shen et al. 2023]. These studies confirm the generality of our approach: just like extending Java types with roles yields an object-oriented choreographic programming language (Choral), doing the same to the λ-calculus supports the development of a theory of functional choreographic programming [Cruz-Filipe et al. 2022, 2023a].

Safety and liveness. The theory of choreographic languages comes with strong safety and liveness properties, which stem from the high-level abstractions provided by choreographies and their projections to distributed code [Montesi 2023]. Under the same assumptions made (sometimes implicitly) in these proofs, Choral promises the traditional safety and liveness properties of choreographic programming languages. We discuss these properties and how they relate to Choral's modularity and flexibility wrt communication middleware. Furthermore, in the end, we report useful references and outline future research for the formalisation of these promises.

Choreographic programming languages guarantee *communication safety*. That is, processes never try to interact by performing incompatible communication actions—if one action is "send," then the other is a "receive," and the type of the sent message is always (a subtype of) the one expected by the receiver [Carbone et al. 2012; Montesi 2023]. Proofs of this result rely on the assumption that communications is reliable (messages are never lost, duplicated, or corrupted) or at least that message exchange primitives adopt suitable best-effort and timeout strategies [Montesi 2023; Montesi and Peressotti 2017]. Furthermore, communication primitives should respect a locality principle: messages should be dispatched to their intended recipients—and vice versa, receiving from a role gives a message that was sent by that role [Giallorenzo et al. 2018; Montesi 2013]. In Choral, like in any previous implementation of choreographic languages, these assumptions form a contract for the implementation of channels. This contract must be manually enforced, or communication methods might be "wrong." A trivial example is a channel for communicating integers that always returns the constant `1` at the receiver. In the future, it would be interesting to explore the formalisation of contracts for channel middleware and libraries and the development of verified implementations.

Another property of choreographic programming—and probably the most known—is *deadlock-freedom by design*: the code compiled from a choreography is deadlock-free, because it is not possible to write mismatched communications in choreographies [Carbone and Montesi 2013; Dalla Preda et al. 2017; Hirsch and Garg 2022; Jongmans and van den Bos 2022; Montesi 2023]. The relevant assumptions, here, are that foreign code (in our case, Java code) used within a choreography always terminates, and that communication never blocks the sender or receiver indefinitely. In the real world, these assumptions usually do not hold, so it is important to adopt timeout and supervision mechanisms to avoid divergence and deal with failures—as exemplified in Sections 2.5 and 3.2.

Under the same assumptions for deadlock-freedom, a choreography that is tail-recursive gives the stronger liveness property of *starvation-freedom* (or livelock-freedom): no role ever gets stuck, or "starves" [Montesi 2023]. (In deadlock-freedom, it is sufficient that some part of the system keeps running [Kobayashi 2000].) The same holds for Choral.

Last, choreographic programming languages typically guarantee *race-freedom*, provided that no role uses the same channel in parallel via internal threads [Carbone et al. 2012; Honda et al. 2016; Lanese et al. 2008; Montesi 2023]—unless the underlying middleware can disambiguate messages, but this is not often the case. Choral gives finer control over race freedom, thanks to its expressive (Channel) types (cf. Section 2.4). For example, given a bidirectional channel, we can pass it as a directed channel in one direction and as a directed channel in the other direction to two separate threads. The API of directed channels would then allow the two threads to use the channel, respectively, only for sending or for receiving, which is safe to do in parallel. This control over channel usage was found to be useful in the choreographic implementation of full-duplex asynchronous communication (a pattern where roles can freely interleave different requests and responses in both directions), as found in the IRC client-server protocol. Lugovic and Montesi [2023] present a detailed discussion of this pattern and an interoperable implementation of IRC in Choral.

Since channel implementations play a key role in the properties discussed above, these implementations should ideally be simple, well-tested, or even verified. Choral's features allow for encapsulating more sophisticated interaction behaviour as choreographies. Lugovic and Montesi [2023] started this activity, offering a reusable Choral library for programming asynchronous reactive protocols.

While, in principle, composing choreographies in Choral preserves safety and liveness, one must pay attention to the interplay with local code. Most notably, local code is free to compose and intertwine multiple instances of the same or different choreographies, as we exemplified in Section 3.2. This flexibility makes it possible to write two local programs that instantiate two choreographies and try to communicate with each other in incompatible orders, share the same channel across different instances of the same choreography running in parallel, and so on. Doing so can create communication errors and make choreographies trivially timeout and fail. This problem can be tackled in several ways, which we leave to future work. Briefly, one option is to devise middleware for making local programs agree on which choreography they want to engage in at any time. One such middleware can avoid some incompatibilities, create dedicated channels, or, in general, throw a runtime exception in case of disagreements. Other potential solutions include devising static or runtime checks for how local code composes instances of choreographies. Options for these checks encompass constraints such as keeping the graph of connections among local programs acyclic; making (projections of) choreographies that run in parallel have dedicated channels; ensuring that call to different methods of different choreographic objects follow compatible orderings; and/or synthesising an overall choreography that describes the collective behaviour of how local programs combine their (sub)choreographies, thereby providing a witness of safety and liveness [Carbone and Montesi 2013; Coppo et al. 2016; Cruz-Filipe et al. 2017; Honda et al. 2016; Jacobs et al. 2022; Lange et al. 2015; Montesi and Yoshida 2013; Voinea et al. 2020]. Some of these methods could be applied via structured concurrency libraries backed by middleware.

"Bad" programming of code that interacts with projections of choreographies is not a new issue, as it was already present in previous implementations of choreographic languages [Carbone and Montesi 2013; Dalla Preda et al. 2017; Montesi 2013; Scalas et al. 2017]. In fact, this is a general issue when composing code from libraries that engage in communications, even if these are not generated from choreographies. At the very least, Choral already guarantees type safety: the APIs of choreography projections are always respected.

Formalising Choral's ideas, implementation, and guarantees is an interesting area of research. As already mentioned, the principles of compiling terms and data types equipped with roles as in Choral have been studied in the simple setting of the λ-calculus [Cruz-Filipe et al. 2022, 2023a; Graversen et al. 2023]. The proofs in these models are not machine-checked. Mechanisation could be achieved by building on the existing formalisations of the foundations of choreographic languages made with theorem provers [Castro-Perez et al. 2021; Cruz-Filipe et al. 2021a, b, 2023; Hirsch and Garg 2022; Pohjola et al. 2022]. These efforts are still focused on choreographic programming languages that are far less expressive than Choral, which by contrast is a large programming language that inherits all the complexities of both choreographic and object orientation.

Selection inference. Choral requires the programmer to insert the necessary selections to achieve knowledge of choice (Section 2.3). Developing techniques for inferring these selections automatically is an ongoing research topic. Typically, these techniques either modify the source choreography to include extra selections or inject hidden communications in the generated endpoint code [Basu and Bultan 2016; Cruz-Filipe and Montesi 2020; Dalla Preda et al. 2017; Jongmans et al. 2015; Lanese et al. 2013]. However, there is no silver bullet.

(1) In general, it is unfeasible to detect automatically what the optimal selection strategy is. This is a problem for both approaches (modifying the source choreography or injecting hidden communications in the generated code). Say that `A` needs to inform `B` and `C` of a choice by using point-to-point channels. Should `A` send the first selection to `B` or to `C`? That might depend on whether `B` has a longer task to perform in reaction to the selection compared to `C`, or vice versa. (Whichever has the longest task to start should get the selection first, to increase parallelism.) Then, what if multiple channels are available? For example, if `A` shares a fast channel with `B` but not with `C`, and `B` shares a fast channel with `C`, then it might be good that `A` informs `B` and subsequently `B` informs `C` (instead of `A` informing `C` directly). These issues become even more sophisticated when considering choreographies with more complex network topologies, scatter-gather channels, recursion, and so on.
(2) If a compiler injects hidden communications in the generated code, then the source choreography program does not faithfully represent the communications enacted by the system any more. This makes the choreography less useful when reasoning about, for example, efficiency—network communications like selections are especially a huge performance factor—and security—hidden extra communications might leak information in ways not intended by the designer of the original protocol.

Since both issues are still the object of active investigation, we decided that the first version of Choral should be a base that future work can use to explore their design spaces. A promising compromise could be a hybrid, assisted way. That is, the programmer should be able to write a choreography including some selections deemed important, but also potentially missing some necessary other selections; then, a tool should detect the missing selections and propose a solution. The programmer could thus decide whether to accept the proposal or improve it manually to achieve their requirements.

In general, we believe that there is a lot of potential in future research on how to optimise communications in Choral. New algorithms might leverage annotations of channels, static analysis, and profiling data. Some algorithms might choose simpler approaches at the expense of parallelism, whereas others might take a more decentralised approach to spreading knowledge of choice to favour parallelism or energy saving (in the Internet of Things, spreading battery consumption evenly or in a focused way might be an advantage depending on the scenario).

Expressivity. We discuss a few interesting directions for future work regarding the expressivity of Choral and its type system. In general, we believe that our choreographic interpretation of object-oriented programming (OOP) allows for importing established techniques from type theory to reason statically about roles in useful ways.

Choral can capture a variety of interaction patterns, like scatter-gather and producers-consumers with races. Nevertheless, there are cases where our types can be coarse. For instance, the following could be an interface of a channel where two receivers, `B` and `C`, race to consume a message.

```
interface RaceDiChannel(A,B,C)<T@X> {
 <S@Y extends T@Y> DPair@(B,C)<Optional<S>,Optional<S>> com(S@A m);
}
```

The return type of method `com` above guarantees that both receivers will have a value of type `Optional<S>` located at them. However, depending on the behaviour that the programmer wishes to model, the interface above could be an over-approximation. For example, implementations that simply discard the message sent from `A` or that deliver the message to both `B` and `C` will satisfy the return type. Currently, we cannot express the type of a channel that forbids such implementations, e.g., a channel guaranteeing that exactly one between `B` and `C` will obtain the message. This means that, in such cases, we have to "pollute" the continuation of the choreography with local checks at both potential receivers.

A way to achieve a more specific type for races could be to extend Choral types with existential quantification over role parameters. For example, we could write

```
S@D with D in [B,C]
```

to express an instance of `S` located at some role `D` in the list `[B,C]` (i.e., `S@D` can be either `S@B` or `S@C`). With this type, we can write a more specific signature: method `com` returns a value of type `S@B` or of type `S@C`, but we cannot statically know which of the two types.

```
interface RaceDiChannel(A,B,C)<T@X> { <S@Y extends T@Y> S@D with D in [B,C] com(S@A m); }
```

Although there is some work on the use of existential quantification in simple choreography languages [Jongmans and Yoshida 2020], its application and integration with a general-purpose language like Choral poses some challenges and design choices. For instance, should roles that lose the race in method `com` be blocked? If so, then is this specific to this method or the standard interpretation of every method with an existential return type? These and similar questions beg for a thorough investigation and go beyond Choral. In fact, a satisfactory and general handling of races in choreographic programming languages is still missing. In addition to the ideas just proposed, useful inspiration to address this aspect might come from nondeterministic choice in choreographies [Lanese et al. 2008; Montesi 2023] and choreographic languages for the verification of message-passing parallel programs [Vasconcelos et al. 2022].

Another limitation of the current type system is that the number of role parameters of a choreography is fixed. This limitation is common to many choreographic languages. [Deniélou and Yoshida 2011] developed a theory for parameterising choreographies over "collections of roles," whose sizes are determined at runtime. All roles in the same collection must be treated uniformly (e.g., broadcast). We can import that feature to Choral by allowing for role parameters to be collections. For example, we could prefix a role parameter declaration with `*`, as in `*Ds`, to specify that it is a collection of roles. Then, we can write the type of a channel for broadcasting data from `A` to all roles in the collection `Ds` as follows.

```
interface BroadcastDiChannel(A, *Ds)<T@X> { /* ... */ }
```

The method `com` for this channel should take a value of type `S@A` and return a value of type `S@D` for every role `D` in the collection `Ds`. This would require investigating how Choral can be extended with types for distributed data collections, as well.

Error handling. Choral supports exception handling at a single role, which can then propagate errors to others via knowledge of choice. However, in our experience, it is more convenient to represent failures in return types, like we did in Section 3.1 by using `Optional`. The channel APIs that we showed in this article are implemented by performing automatic retries. These APIs also have equivalent versions that wrap results in `Result` objects—essentially sum types of the transmitted value type and an error type, as in Go and Rust. Choosing among these implementations is up to the choreography programmer, and programmers might also devise channel implementations with their strategies (e.g., exponential backoff with a bound on the number of retries). Our compiler can, in principle, be extended to synthesise coordination for distributed exceptions, theorised by Carbone et al. [2008].

Type and role inference. Choral's intended audience consists of developers familiar with OOP and, more specifically, Java. For example, our syntax extends that of Java and our library of channels follows common OOP practices, like coding to interfaces [Gamma et al. 1995]. Our design choices rely heavily on generics to achieve reusability, similar to what is done in the standard Java Collections framework [Naftalin and Wadler 2006]. However, this comes at the cost of requiring that programmers have experience with generics and additional parameters in code.

Standard remedies to the verbosity of generics include shorthands like the diamond notation and type inference [Stadelmeier et al. 2022]. We plan to lift these features to Choral and expect that the standard solutions adopted for Java will be applicable in scenarios where roles are known, without major modifications. For example, with such facilities, we could rewrite method `consumeItems` from Section 2.3 by removing all generic instantiations in its body, as follows.

```
consumeItems(DiChannel@(A,B)<Item> ch, Iterator@A<Item> it, Consumer@B<Item> consumer) {
 if (it.hasNext()) {
  ch.select(Choice@A.GO);
  it.next() >> ch::com >> consumer::accept;
  consumeItems(ch, it, consumer);
 } else {
  ch.select(Choice@A.STOP);
 }
}
```
Choral Code

Similar to generics, role parameterisation adds crucial flexibility at the cost of added verbosity. It would therefore be interesting to explore inference mechanisms for role parameters, as well, in order the lighten the syntax of Choral even further. At a basic level, programmers would be able to omit roles when these can be unambiguously determined from the context, e.g., in assignments and some method invocations. The next snippets exemplify the potential gain in simplicity from the current situation (left) to one with this feature (right).

```
// Without inference
List@A<String> l =
  List@A.<String>of("Yes"@A, "No"@A);
String@A x = l.get(0@A);
```

```
// With inference
List@A<String> l =
  List.of("Yes", "No");
var x = l.get(0);
```

We conjecture that this feature can be achieved by machinery similar to that already used for inferring generic parameters, because role information is available from type declarations and the typing context without ambiguity.

At a more advanced level, we could allow programmers to delegate to the Choral compiler decisions on the placement of data and computation. Consider the snippet below.

```
1 int@C m(int@A x, int@B y) {
2  int z = x + y;
3  return z;
4 }
```

Since x and y reside at different roles, the location of z is ambiguous. Even more, performing the addition at Line 2 requires communication, which the compiler would need to infer and inject by appropriately using channels available from the context. This faces similar challenges to the ones previously discussed for selection inference. We believe that exploring methods for the synthesis of communication strategies in choreographies is an interesting research line in general, with a scope that goes beyond that of Choral's.

Other features from Java. There are other features provided by Java (or other object-oriented languages) that Choral could benefit from. We do not discuss them in detail, because we do not expect that lifting them to choreographies would pose significant challenges. For example, we believe that adding wildcard types (?) would be a natural adaptation of Java's mechanism.

The patterns and libraries that support idiomatic Java programming, like streams (from the package `java.util.stream`), are immediately available in Choral thanks to our lifting of Java types to Choral types with a single role. In some cases, it can be interesting to generalise these patterns to multiple roles. For example, a possible interpretation of a "choreographic stream" at two roles A and B could be that of a stream of elements distributed at these roles (`Stream@(A, B)<T@(C, D)>`). Methods for stream operations would then take choreographies at A and B as input. Whether these choreographies would perform communications between the two roles or not would be left to the implementor and is irrelevant to the implementation of the stream.

Asynchronous programming. The choreographies that we presented here use channel APIs as if they were blocking. This does not mean that an endpoint must dedicate a thread for participating in a choreography: future versions of Java will include fibers and the asynchronous execution of blocking APIs (reactor pattern) [OpenJDK 2020]. Choral is compatible with this direction. Should programmers want to program a choreography explicitly for asynchronous execution by using continuation-passing style, our channel APIs can be extended to take choreographic continuations as parameters.

Fluid APIs from choreographic specifications. As we mentioned and discussed in Section 1, previous work explored the automatic generation of fluid APIs that enforce the code to follow a choreographic specification [Scalas et al. 2017]. Such a choreographic language cannot include computation, so it cannot express any of our use cases, and its approach does not support modularity and API control, as we discussed more in detail in the Introduction. Thus, Choral brings two improvements. First, our APIs are more reusable: they change only if the source API is changed, not if the communication behaviour of a method is simply updated. Second, the APIs of our compiled Java code are more idiomatic: they are plain object APIs that look like the typical task-oriented APIs distributed by cloud vendors [Murty 2008; Wilder 2012], which makes Choral a candidate drop-in replacement for current development practices.

Choreography-based verification and testing. Choreographic languages that are less expressive than Choral (e.g., they cannot include computation) have been used also to verify that interactions among objects respect a protocol. This is obtained by statically checking method invocations, either by using typestates [Kouzapas et al. 2018] or model checking [Scalas et al. 2019]. As noted by Hirsch and Garg [2022], choreographic programming offers a simpler development method.

Indeed, verification approaches require the programmer to design both a choreographic specification and then manually take care of writing a correct implementation of the projection of each role. On the contrary, choreographic programming (and hence Choral) generates the latter automatically. Additionally, since our approach is based on compilation instead of verification, we can provide a more expressive choreographic language and apply formal reasoning techniques at the level of choreographies as in, e.g., Cruz-Filipe et al. [2023b]; Jongmans and van den Bos [2022]. Because choreographies syntactically elicit interactions, carrying out verification at this level avoids the cost of reconstructing interactions from endpoint implementations, which usually leads to a combinatorial explosion of cases.

Choreographic languages without computation have been used also in a tool for testing abstract specifications of components given as labelled transition systems [Coto et al. 2021]. The purpose, there, is to test that the communication behaviour of a component (given as a labelled transition system) complies with a choreography. By contrast, our testing tool is the first aimed at testing the functional correctness of a choreography and its generated implementation. Choral's capability of expressing internal computation is important to this end, since it allows us to write arbitrary checks on the distributed state of the system.

8 CONCLUSION

With the increased adoption of cloud computing, edge computing, the Internet of Things, and microservices, the need for libraries that implementors can use to participate correctly in choreographies is growing steadily [Atzori et al. 2010; Dragoni et al. 2017; Murty 2008; Wilder 2012]. Building on previous results on the theory of choreographies, choreographic programming came with the promise of aiding in the implementation of choreography-compliant concurrent and distributed software [Montesi 2015]. While the approach has been successfully applied in principle to different scenarios [Cruz-Filipe and Montesi 2016; Dalla Preda et al. 2017; Lluch-Lafuente et al. 2015; López and Heussen 2017; López et al. 2016], the link between choreographic programming and mainstream programming has remained unexplored until now (all implementations rely on the Jolie programming language [Montesi et al. 2014], which is based on the theory of Calculus of Communicating Systems [Milner 1980]). Among the most important consequences, none of the implementations of the paradigm so far properly supported modularity—generating reusable libraries and controlling their APIs.

In this article, we have taken a fundamental step in the pursuit of the choreographic programming agenda. We have also shown that choreographies can be modelled by extending mainstream abstractions (in our case, objects) and that this leads to a choreographic programming language that supports modularity and can integrate with existing Java code. Choral is sufficiently expressive to capture use cases of different kinds, discussed the design of our compiler, and performed a first evaluation, which points out that the approach is promising. It is thus a step toward equipping programmers with a tool that safely ferries them from the design of choreographies to compliant implementations at the press of the proverbial button.

In the future, Choral could also be a useful vector for the application of research on choreographies based on other paradigms (automata, processes, etc.): researchers could develop translations of their choreography models to Choral, and then leverage our compiler to obtain library implementations that can be used in mainstream software (in Java). Hopefully, this will lead to more implementations of choreography theories, allowing for their evaluation [Ancona et al. 2016].

APPENDIX

A PROJECTION TO JAVA

A.1 Projection

We omit modifiers (MD) and annotations (AN), they are rendered by the projection as they are.

$$(\mathit{Enum}) \quad \llparenthesis \mathtt{enum}\ \mathit{id}@\mathsf{A}\ \{\overline{\mathit{id}}\} \rrparenthesis = \mathtt{enum}\ \mathit{id}\ \{\overline{\mathit{id}}\}$$

$$(\mathit{Interface}) \quad \llparenthesis \mathtt{interface}\ \mathit{id}@(\overline{\mathsf{A}})\ \langle\overline{\mathit{FTP}}\rangle\ \mathtt{extends}\ \mathit{TE}\ \overline{\&\ \mathit{TE}}\ \{\overline{\mathit{MDef};}\} \rrparenthesis =$$
$$\left[\ \mathtt{interface}\ \mathrm{name}(\mathit{id}, \mathsf{A}, \overline{\mathsf{A}})\ \langle\overline{\llparenthesis \mathit{FTP} \rrparenthesis}\rangle\ \mathtt{extends}\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}\ \overline{\&\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}}\ \{\overline{\llparenthesis \mathit{MDef} \rrparenthesis^{\mathsf{A}};}\}\ \middle|\ \mathsf{A} \in \overline{\mathsf{A}}\ \right]$$

$$(\mathit{Class}) \quad \llparenthesis \mathtt{class}\ \mathit{id}@(\overline{\mathsf{A}})\ \langle\overline{\mathit{FTP}}\rangle\ \mathtt{extends}\ \mathit{TE}\ \mathtt{implements}\ \mathit{TE}\overline{,\ \mathit{TE}}\ \{\overline{\mathit{CField}}\ \overline{\mathit{CConst}}\ \overline{\mathit{MDef};}$$
$$\overline{\mathit{MDef}\{\mathit{Stm}\}}\} \rrparenthesis = \Big[\ \mathtt{class}\ \mathrm{name}(\mathit{id}, \mathsf{A}, \overline{\mathsf{A}})\ \mathtt{extends}\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}\ \mathtt{implements}\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}\ \overline{,\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}}$$
$$\{\overline{\llparenthesis \mathit{CField} \rrparenthesis^{\mathsf{A}}}\ \overline{\llparenthesis \mathit{CConst} \rrparenthesis^{\mathsf{A}}}\ \overline{\llparenthesis \mathit{MDef} \rrparenthesis^{\mathsf{A}};}\ \overline{\llparenthesis \mathit{MDef} \rrparenthesis^{\mathsf{A}}\{\llbracket \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} \rrbracket\}}\}\ \Big|\ \mathsf{A} \in \overline{\mathsf{A}}\ \Big]$$

$$(\mathit{FTP}) \quad \llparenthesis \mathit{id}@(\overline{\mathsf{A}})\ \mathtt{extends}\ \mathit{TE}\ \overline{,\ \mathit{TE}} \rrparenthesis = \begin{cases} [\ \mathit{id}\_\mathsf{A}\ \mathtt{extends}\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}\ \overline{\&\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}} \mid \mathsf{A} \in \overline{\mathsf{A}}\ ] & \text{if } |\mathsf{A}| \geq 1 \\ \mathit{id}\ \mathtt{extends}\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}} & \text{otherwise} \end{cases}$$

$$(\mathit{TE}) \quad \llparenthesis \mathtt{void} \rrparenthesis^{\mathsf{A}} = \mathtt{void}$$

$$\llparenthesis \mathit{id}@(\overline{\mathsf{B}})\texttt{<}\overline{\mathit{TE}}\texttt{>} \rrparenthesis^{\mathsf{A}} = \begin{cases} \mathit{id}\texttt{<}\overline{\llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}}\texttt{>} & \overline{\mathsf{B}} = \mathsf{A} \\ \mathit{id}\_\mathit{A}'\texttt{<}\overline{\llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}}\texttt{>} & \mathsf{A} \text{ is the } i\text{-th element of } \overline{\mathsf{B}} \text{ and } \mathtt{roleName}(\mathit{id}, i) = \mathit{A}' \\ \mathtt{Unit} & \text{otherwise} \end{cases}$$

$$(\mathit{MDef}) \quad \llparenthesis \langle\overline{\mathit{FTP}}\rangle\ \mathit{TE}\ \mathit{id}\ (\overline{\mathit{TE}\ \mathit{id}}) \rrparenthesis^{\mathsf{A}} = \langle\overline{\llparenthesis \mathit{FTP} \rrparenthesis}\rangle\ \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}\ \mathit{id}\ (\overline{\llparenthesis \mathit{TE}\ \mathit{id} \rrparenthesis^{\mathsf{A}}})$$

$$(\mathit{CField}) \quad \llparenthesis \overline{\mathit{TE}\ \mathit{id};} \rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}} \mathit{id}; & \text{if } \mathsf{A} \in \mathrm{rolesOf}(\mathit{TE}) \\ [\mathit{blank}] & \text{otherwise} \end{cases}$$

$$(\mathit{CConst}) \quad \llparenthesis \mathit{id}\ (\overline{\mathit{TE}\ \mathit{id}})\{\mathit{Stm}\} \rrparenthesis^{\mathsf{A}} = \mathit{id}\_\mathsf{A}(\overline{\llparenthesis \mathit{TE}\ \mathit{id} \rrparenthesis^{\mathsf{A}}})\{\llbracket \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} \rrbracket\}$$

$$(\mathit{Stm}) \quad \llparenthesis \mathit{nil} \rrparenthesis^{\mathsf{A}} = \llparenthesis [\mathit{blank}] \rrparenthesis^{\mathsf{A}} = [\mathit{blank}]$$

$$\llparenthesis \mathtt{return}\ \mathit{Exp}; \rrparenthesis^{\mathsf{A}} = \mathtt{return}\ \llparenthesis \mathit{Exp} \rrparenthesis^{\mathsf{A}};$$

$$\llparenthesis \mathit{Exp}; \mathit{Stm} \rrparenthesis^{\mathsf{A}} = \begin{cases} \mathtt{switch}(\llparenthesis \mathit{Exp} \rrparenthesis^{\mathsf{A}})\{ & \text{if } \mathrm{typeOf}(\mathit{Exp}) \mathrel{\texttt{<:}} \mathtt{Enum@A}, \\ \quad \mathtt{case}\ \mathit{id}_3\texttt{->}\{\llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}}\} & \quad \mathit{Exp} = \overline{\mathit{Exp}'}.\langle\overline{\mathit{TE}}\rangle \mathit{id}_1(\mathit{id}_2@\mathsf{A}'.\mathit{id}_3) \text{ and} \\ \quad \mathtt{default}\texttt{->}\{\mathtt{throw}\ \ldots\}\} & \quad \mathtt{@SelectionMethod} \in \mathtt{annotOf}(\mathit{id}_1) \\ \llparenthesis \mathit{Exp} \rrparenthesis^{\mathsf{A}}; \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{if } \mathsf{A} \in \mathrm{rolesOf}(\mathit{Exp}) \\ \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{otherwise} \end{cases}$$

$$\llparenthesis \mathit{TE}\ \mathit{id}\texttt{=}\mathit{Exp}; \mathit{Stm} \rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis \mathit{TE} \rrparenthesis^{\mathsf{A}}\texttt{=}\llparenthesis \mathit{Exp} \rrparenthesis^{\mathsf{A}}; \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{if } \mathsf{A} \in \mathrm{rolesOf}(\mathit{TE}) \\ \llparenthesis \mathit{Exp} \rrparenthesis^{\mathsf{A}}; \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{if } \mathsf{A} \in \mathrm{rolesOf}(\mathit{Exp}) \setminus \mathrm{rolesOf}(\mathit{TE}) \\ \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{otherwise} \end{cases}$$

$$\llparenthesis \mathit{Exp}_1\ \mathit{AsgOp}\ \mathit{Exp}_2; \mathit{Stm} \rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis \mathit{Exp}_1 \rrparenthesis^{\mathsf{A}} \mathit{AsgOp} \llparenthesis \mathit{Exp}_2 \rrparenthesis^{\mathsf{A}}; \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{if } \mathsf{A} \in \mathrm{rolesOf}(\mathrm{typeOf}(\mathit{Exp})) \\ \llparenthesis \mathit{Exp}_1 \rrparenthesis^{\mathsf{A}}.\mathtt{id}(\llparenthesis \mathit{Exp}_2 \rrparenthesis^{\mathsf{A}}); \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{if } \mathsf{A} \in \mathrm{rolesOf}(\mathit{Exp}_1, \mathit{Exp}_2) \\ \llparenthesis \mathit{Stm} \rrparenthesis^{\mathsf{A}} & \text{otherwise} \end{cases}$$

$$\llparenthesis \mathtt{if}(\mathit{Exp})\{\mathit{Stm}_1\}\mathtt{else}\{\mathit{Stm}_2\}\mathit{Stm} \rrparenthesis^{\mathsf{A}} =$$

$$\begin{cases} \mathtt{if}(\llparenthesis Exp \rrparenthesis^{\mathsf{A}})\{\llparenthesis Stm_1\rrparenthesis^{\mathsf{A}}\}\mathtt{else}\{\llparenthesis Stm_2\rrparenthesis^{\mathsf{A}}\}\llparenthesis Stm\rrparenthesis^{\mathsf{A}} & \text{if typeOf}(Exp) = \mathtt{boolean@A} \\ \llparenthesis Exp\rrparenthesis^{\mathsf{A}}; \left\{ [\![\llparenthesis Stm_1\rrparenthesis^{\mathsf{A}}]\!] \sqcup [\![\llparenthesis Stm_2\rrparenthesis^{\mathsf{A}}]\!] \right\} \llparenthesis Stm\rrparenthesis^{\mathsf{A}} & \text{otherwise}\end{cases}$$

$$\llparenthesis \{Stm_1\}Stm_2\rrparenthesis^{\mathsf{A}} = \{\llparenthesis Stm_1\rrparenthesis^{\mathsf{A}}\}\llparenthesis Stm_2\rrparenthesis^{\mathsf{A}}$$

$$\llparenthesis \mathtt{try}\{Stm\}\overline{\mathtt{catch}(TE\ id)\{Stm\}}\ Stm\rrparenthesis^{\mathsf{A}} = \mathtt{try}\{\llparenthesis Stm\rrparenthesis^{\mathsf{A}}\}\overline{\llparenthesis \mathtt{catch}(TE\ id)\{Stm\}\rrparenthesis^{\mathsf{A}}}\ \llparenthesis Stm\rrparenthesis^{\mathsf{A}}$$

$$\llparenthesis \mathtt{catch}(TE\ id)\{Stm\}\rrparenthesis^{\mathsf{A}} = \begin{cases} \mathtt{catch}(\llparenthesis TE\rrparenthesis^{\mathsf{A}}\ id)\{\llparenthesis Stm\rrparenthesis^{\mathsf{A}}\} & \text{if } \mathsf{A} \in \text{rolesOf}(TE) \\ [blank] & \text{otherwise}\end{cases}$$

$$(Exp) \qquad \llparenthesis lit\rrparenthesis^{\mathsf{A}} = \begin{cases} l & \text{if } lit = l@(\overline{\mathsf{B}}) \text{ and } \mathsf{A} \in \overline{\mathsf{B}} \\ \mathtt{Unit.id} & \text{otherwise}\end{cases}$$

$$\llparenthesis Exp\ BinOp\ Exp'\rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis Exp\rrparenthesis^{\mathsf{A}} BinOp\ \llparenthesis Exp'\rrparenthesis^{\mathsf{A}} & \text{if } \begin{pmatrix} BinOp \in \{\&\&, ||\} \\ \wedge \text{ rolesOf}(Exp') = \{\mathsf{A}\} \end{pmatrix} \\ & \vee \begin{pmatrix} BinOp \notin \{\&\&, ||\} \\ \wedge\ \mathsf{A} \in \text{rolesOf}(\text{typeOf}(Exp)) \end{pmatrix} \\ \mathtt{Unit.id}(\llparenthesis Exp\rrparenthesis^{\mathsf{A}}, \llparenthesis Exp'\rrparenthesis^{\mathsf{A}}) & \text{if } \begin{pmatrix} BinOp \in \{\&\&, ||\} \\ \wedge \text{ rolesOf}(Exp') = \{\mathsf{A'}\} \end{pmatrix} \\ & \vee BinOp \notin \{\&\&, ||\} \end{cases}$$

let head$(Exp.Exp') = Exp_h$, rest$(Exp.Exp') = Exp_r$

$$\llparenthesis Exp.Exp'\rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis Exp_h\rrparenthesis^{\mathsf{A}}.\mathtt{id}(\llparenthesis Exp_r\rrparenthesis^{\mathsf{A}}) & \text{if } \llparenthesis Exp_h\rrparenthesis^{\mathsf{A}} \in \{\mathtt{Unit.}\bullet\} \\ \mathtt{Unit.id}(\llparenthesis Exp_h\rrparenthesis^{\mathsf{A}}).\overline{Exp_1} & \text{if } \llparenthesis Exp_r\rrparenthesis^{\mathsf{A}} \in \{\mathtt{Unit.}\overline{Exp_1}\} \\ \llparenthesis Exp_h\rrparenthesis^{\mathsf{A}}.\llparenthesis Exp_r\rrparenthesis^{\mathsf{A}} & \text{otherwise}\end{cases}$$

$$\llparenthesis \langle\overline{TE}\rangle id(\overline{Exp})\rrparenthesis^{\mathsf{A}} = \begin{cases} \langle\overline{\llparenthesis TE\rrparenthesis^{\mathsf{A}}}\rangle id(\overline{\llparenthesis Exp\rrparenthesis^{\mathsf{A}}}) & \text{if } \mathsf{A} \in \text{rolesOf}(\text{typeOf}(\langle\overline{TE}\rangle id(\overline{Exp}))) \\ \mathtt{Unit.id}(\overline{\llparenthesis Exp\rrparenthesis^{\mathsf{A}}}) & \text{otherwise}\end{cases}$$

$$\llparenthesis id@(\overline{\mathsf{B}}).\langle\overline{TE}\rangle id(\overline{Exp})\rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis id@(\overline{\mathsf{B}})\rrparenthesis^{\mathsf{A}}.\langle\overline{\llparenthesis TE\rrparenthesis^{\mathsf{A}}}\rangle id(\overline{\llparenthesis Exp\rrparenthesis^{\mathsf{A}}}) & \mathsf{A} \in \overline{\mathsf{B}} \\ \mathtt{Unit.id}(\overline{\llparenthesis Exp\rrparenthesis^{\mathsf{A}}}) & \text{otherwise}\end{cases}$$

$$\llparenthesis \mathtt{new}\ \langle\overline{TE}\rangle id\langle\overline{TE}\rangle(\overline{Exp})\rrparenthesis^{\mathsf{A}} = \begin{cases} \mathtt{new}\ \langle\overline{\llparenthesis TE\rrparenthesis^{\mathsf{A}}}\rangle\llparenthesis id@(\overline{\mathsf{B}})\langle\overline{TE}\rangle\rrparenthesis^{\mathsf{A}}(\overline{\llparenthesis Exp\rrparenthesis^{\mathsf{A}}}) & \mathsf{A} \in \overline{\mathsf{B}} \\ \mathtt{Unit.id}(\overline{\llparenthesis Exp\rrparenthesis^{\mathsf{A}}}) & \text{otherwise}\end{cases}$$

$$(FAcc) \qquad \llparenthesis id\rrparenthesis^{\mathsf{A}} = \begin{cases} id & \mathsf{A} \in \text{rolesOf}(id) \\ \mathtt{Unit.id} & \text{otherwise}\end{cases}$$

$$\llparenthesis id@(\overline{\mathsf{B}}).f\rrparenthesis^{\mathsf{A}} = \begin{cases} \llparenthesis id@(\overline{\mathsf{B}})\rrparenthesis^{\mathsf{A}}.f & \mathsf{A} \in \text{rolesOf}(f) \\ \mathtt{Unit.id} & \text{otherwise}\end{cases}$$

A.2 Merging

$$\dot{\bigsqcup}\ \overline{Stm} = \dot{\bigsqcup}(Stm_1, \cdots, Stm_n) = [\![Stm_1]\!] \sqcup \cdots \sqcup [\![Stm_n]\!]$$

Statements

$$\mathtt{return}\ Exp \sqcup \mathtt{return}\ Exp' = \mathtt{return}\ Exp \sqcup Exp'$$

$$TE\ id; Stm \sqcup TE\ id; Stm' = TE\ id; (Stm \sqcup Stm')$$

$$(Exp_1\ AsgOp\ Exp_2; Stm) \sqcup (Exp'_1\ AsgOp\ Exp'_2; Stm')$$

$$= (Exp_1 \sqcup Exp'_1)\ AsgOp\ (Exp_2 \sqcup Exp'_2); (Stm \sqcup Stm')$$

$$(Exp; Stm) \sqcup (Exp'; Stm') = (Exp \sqcup Exp'); (Stm \sqcup Stm')$$

$$\{Stm_1\}\ Stm_2 \sqcup \{Stm'_1\}\ Stm'_2 = \{Stm_1 \sqcup Stm'_1\}\ (Stm_2 \sqcup Stm'_2)$$

$$\mathtt{if}(Exp)\{Stm_1\}\mathtt{else}\{Stm_2\}Stm \sqcup \mathtt{if}(Exp')\{Stm'_1\}\mathtt{else}\{Stm'_2\}Stm'$$

$$= \mathtt{if}(Exp \sqcup Exp')\{Stm_1 \sqcup Stm'_1\}\mathtt{else}\{Stm_2 \sqcup Stm'_2\}(Stm \sqcup Stm')$$

$$\begin{array}{l}\mathtt{switch}\ (Exp)\{\\ \quad \mathtt{case}\ id_a\texttt{->}\{Stm_a\}\\ \quad \ldots\\ \quad \underline{\mathtt{case}\ id_x\texttt{->}\{Stm_x\}}\\ \quad \mathtt{case}\ id_y\texttt{->}\{Stm_y\}\\ \quad \mathtt{default}\texttt{->}\{Stm_{d1}\}\\ \}\ Stm\end{array} \sqcup \begin{array}{l}\mathtt{switch}\ (Exp)\{\\ \quad \mathtt{case}\ id_a\texttt{->}\{Stm'_a\}\\ \quad \ldots\\ \quad \underline{\mathtt{case}\ id_x\texttt{->}\{Stm'_x\}}\\ \quad \mathtt{case}\ id_z\texttt{->}\{Stm_z\}\\ \quad \mathtt{default}\texttt{->}\{Stm_{d2}\}\\ \}\ Stm'\end{array} = \begin{array}{l}\mathtt{switch}\ (Exp \sqcup Exp')\{\\ \quad \mathtt{case}\ id_a\texttt{->}\{Stm_a \sqcup Stm'_a\}\\ \quad \ldots\\ \quad \underline{\mathtt{case}\ id_x\texttt{->}\{Stm_x \sqcup Stm'_x\}}\\ \quad \underline{\mathtt{case}\ id_y\texttt{->}\{Stm_y\}}\\ \quad \mathtt{case}\ id_z\texttt{->}\{Stm_z\}\\ \quad \mathtt{default}\texttt{->}\{Stm_{d1} \sqcup Stm_{d2}\}\\ \}\ Stm \sqcup Stm'\end{array}$$

$$\mathtt{try}\ \{Stm_1\}\ \overline{\mathtt{catch}\ (TE\ id)\ \{Stm\}}\ Stm_2 \sqcup \mathtt{try}\ \{Stm_3\}\ \overline{\mathtt{catch}\ (TE\ id)\ \{Stm'\}}\ Stm_4$$
$$= \mathtt{try}\ \{Stm_1 \sqcup Stm_3\}\ \overline{\mathtt{catch}\ (TE\ id)\ \{Stm \sqcup Stm'\}}\ Stm_2 \sqcup Stm_4$$

Expressions

let $\bullet \in \{nil, [blank], \mathtt{null}, \mathtt{this}, \mathtt{super}, id\},\ \bullet \sqcup \bullet = \bullet$

let $\bullet \in \{\mathtt{new}\ id\ \langle\overline{TE}\rangle, id\ \langle\overline{TE}\rangle\ \},\ \bullet\ (\overline{Exp}) \sqcup \bullet\ (\overline{Exp'}) = \bullet\ (\overline{Exp \sqcup Exp'})$

$$(Exp_1\ BinOp\ Exp_2) \sqcup (Exp'_1\ BinOp\ Exp'_2;) = (Exp_1 \sqcup Exp'_1)\ BinOp\ (Exp_2 \sqcup Exp'_2)$$

$$Exp_1.Exp_2 \sqcup Exp_3.Exp_4 = (Exp_1 \sqcup Exp_3)(.Exp_2 \sqcup .Exp_4)$$

$$.id \sqcup .id = .id \qquad .id\langle\overline{TE}\rangle(\overline{Exp}) \sqcup .id\langle\overline{TE}\rangle(\overline{Exp'}) = .id\langle\overline{TE}\rangle(\overline{Exp \sqcup Exp'})$$

$$.Exp_1.Exp_2 \sqcup .Exp_3.Exp_4 = (.Exp_1 \sqcup .Exp_3)(.Exp_2 \sqcup .Exp_4)$$

A.3 Normaliser

$$\llbracket nil \rrbracket = nil \qquad \llbracket \mathtt{return}\cdot Exp;\ \rrbracket = \mathtt{return}\cdot\llbracket Exp \rrbracket; \qquad \llbracket TE\cdot id; Stm \rrbracket = TE\cdot id; \llbracket Stm \rrbracket$$

$$\llbracket Exp\cdot AsgOp\cdot Exp'; Stm \rrbracket = \llbracket Exp \rrbracket\cdot AsgOp\cdot\llbracket Exp' \rrbracket; \llbracket Stm \rrbracket$$

$$\llbracket \{Stm\}\cdot Stm \rrbracket = \{\llbracket Stm \rrbracket\}\cdot\llbracket Stm \rrbracket$$

$$\textsc{noop}(Exp) = \begin{cases} [blank] & \text{if } Exp \in \{\mathtt{Unit.id},\ id\ \overline{.id}, \mathtt{this}, \mathtt{null}\} \\ Exp & \text{otherwise} \end{cases}$$

$$\llbracket Exp; Stm \rrbracket = \begin{cases} \llbracket Stm \rrbracket & \text{if } \textsc{noop}(\llbracket Exp \rrbracket) = [blank] \\ \llbracket Exp \rrbracket; \llbracket Stm \rrbracket & \text{otherwise} \end{cases}$$

$$\llbracket \mathtt{if}(Exp)\{Stm_1\}\mathtt{else}\{Stm_2\}Stm \rrbracket = \mathtt{if}(\llbracket Exp \rrbracket)\{\llbracket Stm_1 \rrbracket\}\mathtt{else}\{\llbracket Stm_2 \rrbracket\}\llbracket Stm \rrbracket$$

$$\llbracket \mathtt{switch}(Exp)\{\overline{\mathtt{case}\cdot id\texttt{->}\{Stm\}}\cdot\mathtt{default}\texttt{->}Stm'\}\cdot Stm \rrbracket$$

$$= \mathtt{switch}(\llbracket Exp \rrbracket)\{\overline{\mathtt{case}\cdot id\texttt{->}\{\llbracket Stm \rrbracket\}}\cdot\mathtt{default}\texttt{->}\llbracket Stm' \rrbracket\}\cdot\llbracket Stm \rrbracket$$

$$\llbracket \mathtt{try}\cdot\{Stm\}\cdot\overline{\mathtt{catch}\cdot(TE\cdot id)\cdot\{Stm\}}\cdot Stm \rrbracket = \mathtt{try}\cdot\{\llbracket Stm \rrbracket\}\cdot\overline{\mathtt{catch}\cdot(TE\cdot id)\cdot\{\llbracket Stm \rrbracket\}}\cdot\llbracket Stm \rrbracket$$

Expressions

$$\llbracket \mathtt{null} \rrbracket = \mathtt{null} \qquad \llbracket \mathtt{null} \rrbracket^\star = \langle \text{false}, \mathtt{null} \rangle \qquad \llbracket \mathtt{this} \rrbracket = \mathtt{this} \qquad \llbracket \mathtt{this} \rrbracket^\star = \langle \text{false}, \mathtt{this} \rangle$$

$$\llbracket id \rrbracket = id \qquad \llbracket id \rrbracket^\star = \langle \text{false}, id \rangle \qquad \text{let } \llbracket id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket^\star = \langle \bullet, \diamond \rangle,\ \llbracket id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket = \diamond$$

$$\text{let } \overline{\llbracket Exp \rrbracket^\star} = \overline{\langle \bullet, \diamond \rangle},\ \llbracket id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket^\star = \langle \bigvee \overline{\bullet}, id\langle\overline{TE}\rangle(\overline{\diamond}) \rangle$$

$$\text{let } \llbracket \mathtt{new}\cdot id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket^\star = \langle \bullet, \diamond \rangle,\ \llbracket \mathtt{new}\cdot id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket = \diamond$$

$$\text{let } \overline{\llbracket Exp \rrbracket^\star} = \overline{\langle \bullet, \diamond \rangle},\ \llbracket \mathtt{new}\cdot id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket^\star = \langle \bigvee \overline{\bullet}, \mathtt{new}\cdot id\langle\overline{TE}\rangle(\overline{\diamond}) \rangle$$

$$\llbracket Exp\cdot BinOp\cdot Exp' \rrbracket = \llbracket Exp \rrbracket\cdot BinOp\cdot\llbracket Exp' \rrbracket$$

$$\text{let } \llbracket Exp.Exp' \rrbracket^\star = \langle \bullet, \diamond \rangle,\ \llbracket Exp.Exp' \rrbracket = \begin{cases} \llbracket \diamond \rrbracket & \text{if } \bullet = \text{true} \\ \diamond & \text{otherwise} \end{cases}$$

$$\llbracket Exp.Exp' \rrbracket^\star = \begin{cases} \langle \text{true}, \mathtt{Unit.id}(\overline{Exp}) \rangle & \text{if } Exp.Exp' = \mathtt{Unit.id.id}(\overline{Exp}) \\ \langle \text{true}, Exp \rangle & \text{if } Exp.Exp' = \mathtt{Unit.id}(Exp) \\ \langle \text{false}, \mathtt{Unit.id} \rangle & \text{if } Exp = \mathtt{Unit} \text{ and } \llbracket .Exp' \rrbracket^\star = \langle \bullet, [blank] \rangle \\ \langle \bullet \vee \bullet', \diamond\ \diamond' \rangle & \text{otherwise, let } \llbracket Exp \rrbracket^\star = \langle \bullet, \diamond \rangle \\ & \ \text{and } \llbracket .Exp' \rrbracket^\star = \langle \bullet', \diamond' \rangle \end{cases}$$

$$\llbracket .id \rrbracket^\star = \langle \text{false}, .id \rangle \qquad \text{let } \llbracket .Exp \rrbracket^\star = \langle \bullet, \diamond \rangle,\ \llbracket .id.Exp \rrbracket^\star = \langle \bullet, .id\ \diamond \rangle$$

$$\text{let } \overline{\llbracket Exp \rrbracket^\star} = \overline{\langle \bullet, \diamond \rangle},\ \llbracket .id(\overline{Exp}) \rrbracket^\star = \begin{cases} \langle \bigvee \overline{\bullet}, .id(\overline{\diamond}) \rangle & \text{if } .id \neq \mathtt{.id} \\ \langle \text{true}, [blank] \rangle & \text{if } \overline{\textsc{noop}(\diamond)} = \overline{[blank]} \\ \langle \bigvee \overline{\bullet} \vee |\overline{\diamond}| \neq |\overline{\star}|, \mathtt{.id}(\star) \rangle & \text{otherwise, let } \overline{\textsc{noop}(\diamond)} = \overline{\star} \end{cases}$$

$$\text{let } \llbracket .id\langle\overline{TE}\rangle(\overline{Exp}) \rrbracket^\star = \langle \bullet, \diamond \rangle \text{ and } \llbracket .Exp \rrbracket^\star = \langle \bullet', \diamond' \rangle,$$

$$\llbracket .id\langle\overline{TE}\rangle(\overline{Exp}).Exp \rrbracket^\star = \langle \bullet \vee \bullet', \diamond\ \diamond' \rangle$$

REFERENCES

Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2000. Inference of message sequence charts. In *Proceedings of the 22nd International Conference on on Software Engineering (ICSE'00)*, Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf (Eds.). ACM, 304–313. https://doi.org/10.1145/337180.337215

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, 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. 2016. Behavioral types in programming languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The internet of things: A survey. *Comput. Netw.* 54, 15 (2010), 2787–2805.

Marco Autili, Paola Inverardi, and Massimo Tivoli. 2018. Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. *Sci. Comput. Program.* 160 (2018), 3–29. https://doi.org/10.1016/j.scico.2017.10.010

Samik Basu and Tevfik Bultan. 2016. Automated Choreography repair. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE'16), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS'16) (Lecture Notes in Computer Science)*, Perdita Stevens and Andrzej Wasowski (Eds.), Vol. 9633. Springer, 13–30. https://doi.org/10.1007/978-3-662-49665-7_2

Samik Basu, Tevfik Bultan, and Meriem Ouederni. 2012. Deciding Choreography realizability. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, John Field and Michael Hicks (Eds.). ACM, 191–202. https://doi.org/10.1145/2103656.2103680

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

Mario Bravetti and Gianluigi Zavattaro. 2007. Towards a unifying theory for choreography conformance and contract compliance. In *Proceedings of the 6th International Symposium on Software Composition (SC'07) (Lecture Notes in Computer Science)*, Markus Lumpe and Wim Vanderperren (Eds.), Vol. 4829. Springer, 34–50. https://doi.org/10.1007/978-3-540-77351-1_4

Andrea Burattin. 2016. PLG2: Multiperspective process randomization with online and offline simulations. In *Proceedings of the BPM Demo Track Co-located with the 14th International Conference on Business Process Management (BPM'16) (CEUR Workshop Proceedings)*, Leonardo Azevedo and Cristina Cabanillas (Eds.), Vol. 1789. CEUR-WS.org, 1–6. Retrieved from https://ceur-ws.org/Vol-1789/bpm-demo-2016-paper1.pdf

Andrea Burattin and Alessandro Sperduti. 2011. PLG: A framework for the generation of business process models and their execution logs. In *Proceedings of the International Workshops and Education Track on Business Process Management (BPM'10)*. Springer, 214–219.

Marco Carbone. 2009. Session-based choreography with exceptions. *Electron. Notes Theor. Comput. Sci.* 241 (2009), 35–55. https://doi.org/10.1016/j.entcs.2009.06.003

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured interactional exceptions in session types. In *CONCUR Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08) (Lecture Notes in Computer Science)*, Franck van Breugel and Marsha Chechik (Eds.), Vol. 5201. Springer, 402–417. https://doi.org/10.1007/978-3-540-85361-9_32

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.* 34, 2 (2012), 8:1–8:78. https://doi.org/10.1145/2220365.2220367

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 263–274. https://doi.org/10.1145/2429069.2429101

Luca Cardelli and Andrew D. Gordon. 2000. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 365–377. https://doi.org/10.1145/325694.325742

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On global types and multi-party sessions. In *Formal Techniques for Distributed Systems*. Springer, 1–28.

David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for certified multiparty computation: From mechanised metatheory to certified multiparty processes. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 237–251. https://doi.org/10.1145/3453483.3454041

David Castro-Perez and Nobuko Yoshida. 2020. Compiling first-order functions to session-typed parallel code. In *Proceedings of the 29th International Conference on Compiler Construction (CC'20)*, Louis-Noël Pouchet and Alexandra Jimborean (Eds.). ACM, 143–154. https://doi.org/10.1145/3377555.3377889

Tzu-Chun Chen and Kohei Honda. 2012. Specifying stateful asynchronous properties for distributed programs. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12) (Lecture Notes in Computer Science)*, Maciej Koutny and Irek Ulidowski (Eds.), Vol. 7454. Springer, 209–224. https://doi.org/10.1007/978-3-642-32940-1_16

Choral Development Team. 2020. Choral Language Website. Retrieved from https://www.choral-lang.org

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO'06) (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4709. Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

Alex Coto, Roberto Guanciale, and Emilio Tuosto. 2021. An abstract framework for choreographic testing. *J. Log. Algebraic Methods Program.* 123 (2021), 100712. https://doi.org/10.1016/j.jlamp.2021.100712

Luís Cruz-Filipe, Eva Graversen, Lovro Lugovic, Fabrizio Montesi, and Marco Peressotti. 2022. Functional choreographic programming. In *Proceedings of the 19th International Colloquium on Theoretical Aspects of Computing (ICTAC'22) (Lecture Notes in Computer Science)*, Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu (Eds.), Vol. 13572. Springer, 212–237. https://doi.org/10.1007/978-3-031-17715-6_15

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2023a. Modular compilation for higher-order functional choreographies. In *Proceedings of the 37th European Conference on Object-Oriented Programming (ECOOP'23) (Leibniz International Proceedings in Informatics (LIPIcs))*, Karim Ali and Guido Salvaneschi (Eds.), Vol. 263. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:37. https://doi.org/10.4230/LIPIcs.ECOOP.2023.7

Luís Cruz-Filipe, Eva Graversen, Fabrizio Montesi, and Marco Peressotti. 2023b. Reasoning about choreographic programs. In *Proceedings of the Conference on Coordination Models and Languages (Lecture Notes in Computer Science)*, Sung-Shik Jongmans and Antónia Lopes (Eds.), Vol. 13908. Springer, 144–162. https://doi.org/10.1007/978-3-031-35361-1_8

Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. 2017. The paths to choreography extraction. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'17), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS'17) (Lecture Notes in Computer Science)*, Javier Esparza and Andrzej S. Murawski (Eds.), Vol. 10203. 424–440. https://doi.org/10.1007/978-3-662-54458-7_25

Luís Cruz-Filipe, Kim S. Larsen, Fabrizio Montesi, and Larisa Safina. 2022. Implementing choreography extraction. Retrieved from https://arXiv:2205.02636

Luís Cruz-Filipe and Fabrizio Montesi. 2016. Choreographies in practice. In *Proceedings of the 36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'16), held as part of the 11th International Federated Conference on Distributed Computing Techniques (DisCoTec'16) (Lecture Notes in Computer Science)*, Elvira Albert and Ivan Lanese (Eds.), Vol. 9688. Springer, 114–123. https://doi.org/10.1007/978-3-319-39570-8_8

Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theor. Comput. Sci.* 802 (2020), 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021a. Certifying choreography compilation. In *Proceedings of the 18th International Colloquium on Theoretical Aspects of Computing (ICTAC'21) (Lecture Notes in Computer Science)*, Antonio Cerone and Peter Csaba Ölveczky (Eds.), Vol. 12819. Springer, 115–133. https://doi.org/10.1007/978-3-030-85315-0_8

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021b. Formalising a turing-complete choreographic language in coq. In *Proceedings of the 12th International Conference on Interactive Theorem Proving (ITP'21) (LIPIcs)*, Liron Cohen and Cezary Kaliszyk (Eds.), Vol. 193. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 15:1–15:18. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2023. A formal theory of choreographic programming. *J. Autom. Reason.* 67, 21 (2023), 1–34. https://doi.org/10.1007/s10817-023-09665-3

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2017. Dynamic choreographies: Theory and implementation. *Logic. Methods Comput. Sci.* 13, 2 (2017).

Romain Demangeon and Kohei Honda. 2012. Nested protocols in session types. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12) (Lecture Notes in Computer Science)*, Maciej Koutny and Irek Ulidowski (Eds.), Vol. 7454. Springer, 272–286. https://doi.org/10.1007/978-3-642-32940-1_20

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 435–446. https://doi.org/10.1145/1926385.1926435

Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Trans. Inf. Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: Session types without tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

Vincent Massol and Ted Husted. 2004. *Junit in action. Manning.*

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. 2018. Applied choreographies. In *Proceedings of the 38th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'18), held as part of the 13th International Federated Conference on Distributed Computing Techniques (DisCoTec'18) (Lecture Notes in Computer Science)*, Christel Baier and Luís Caires (Eds.), Vol. 10854. Springer, 21–40. https://doi.org/10.1007/978-3-319-92612-4_2

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Choreographies as objects. Retrieved from https://arxiv.org/abs/2005.09520v1

Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty languages: The choreographic and multitier cases. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP'21) (Leibniz International Proceedings in Informatics (LIPIcs))*, Anders Møller and Manu Sridharan (Eds.), Vol. 194. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:27. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22

Saverio Giallorenzo, Fabrizio Montesi, Larisa Safina, and Stefano Pio Zingaro. 2019. Ephemeral data handling in microservices. In *Proceedings of the IEEE International Conference on Services Computing (SCC'19)*, Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama (Eds.). IEEE, 234–236. https://doi.org/10.1109/SCC.2019.00048

Paul A. Grassi, James L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Yee-Yin Choong, et al. 2017. NIST special publication 800-63b: Digital identity guidelines. *Enrollment and Identity Proofing Requirements.* Retrieved from https://pages.nist.gov/800-63-3/sp800-63a.html

Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2023. Alice or Bob?: Process polymorphism in choreographies. Retrieved from https://arXiv:2303.04678

Martin Grotzke. May 2020. Kryo. Retrieved from https://github.com/EsotericSoftware/kryo

Paul Hamill. 2004. *Unit Test Frameworks: Tools for High-quality Software Development.* O'Reilly Media.

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

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

Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. 2008. MQTT-SA publish/subscribe protocol for Wireless Sensor Networks. In *Proceedings of the 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08).* IEEE, 791–798.

Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of session types and behavioural contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. https://doi.org/10.1145/2873052

Intl. Telecommunication Union. 1996. Recommendation Z.120: Message Sequence Chart. https://www.itu.int/rec/T-REC-Z.120-201102-I/en

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Connectivity graphs: A method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.* 6 (2022), 1–33. https://doi.org/10.1145/3498662

Sung-Shik Jongmans and Petra van den Bos. 2022. A predicate transformer for choreographies—Computing preconditions in choreographic programming. In *Proceedings of the 31st European Symposium on Programming (ESOP'22), held as part of the European Joint Conferences on Theory and Practice of Software, (ETAPS'22) (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.), Vol. 13240. Springer, 520–547. https://doi.org/10.1007/978-3-030-99336-8_19

Sung-Shik Jongmans and Nobuko Yoshida. 2020. Exploring type-level bisimilarity towards more expressive multiparty session types. In *Proceedings of the 29th European Symposium on Programming (ESOP'20), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS'20) (Lecture Notes in Computer Science)*, Peter Müller (Ed.), Vol. 12075. Springer, 251–279. https://doi.org/10.1007/978-3-030-44914-8_10

Sung-Shik T. Q. Jongmans, Francesco Santini, and Farhad Arbab. 2015. Partially distributed coordination with Reo and constraint automata. *Serv. Orient. Comput. Appl.* 9, 3-4 (2015), 311–339. https://doi.org/10.1007/s11761-015-0177-y

Anatolii Alekseevich Karatsuba and Yu P. Ofman. 1962. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, Vol. 145. Russian Academy of Sciences, 293–294.

Donald Knuth. 1998. Section 5.2.4: Sorting by merging. *Art Comput. Program.* 3 (1998), 158–168.

Naoki Kobayashi. 2000. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *Proceedings of the International Conference on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics (IFIP TCS'00) (Lecture Notes in Computer Science)*, Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito (Eds.), Vol. 1872. Springer, 365–389. https://doi.org/10.1007/3-540-44929-9_27

Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. Bridging the gap between interaction- and process-oriented choreographies. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08)*, Antonio Cerone and Stefan Gruner (Eds.). IEEE Computer Society, 323–332. https://doi.org/10.1109/SEFM.2008.11

Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2013. Amending choreographies. In *Proceedings of the 9th International Workshop on Automated Specification and Verification of Web Systems (WWV'13)*, António Ravara and Josep Silva (Eds.), Vol. 123. 34–48. https://doi.org/10.4204/EPTCS.123.5

Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From communicating machines to graphical choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM, 221–232. https://doi.org/10.1145/2676726.2676964

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. 517–530.

Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 321–334. https://doi.org/10.1145/1629575.1629606

Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. 2015. Discretionary information flow control for interaction-oriented specifications. In *Proceedings of the Conference on Logic, Rewriting, and Concurrency (Lecture Notes in Computer Science)*, Vol. 9200. Springer, 427–450.

Hugo A. López and Kai Heussen. 2017. Choreographing cyber-physical distributed control systems for the energy sector. In *Proceedings of the Symposium on Applied Computing (SAC'17)*. ACM, 437–443.

Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. 2016. Enforcing availability in failure-aware communicating systems. In *Proceedings of the Conference on Formal Techniques for Networked and Distributed Systems (FORTE'16) (Lecture Notes in Computer Science)*, Vol. 9688. Springer, 195–211.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. 329–339.

Lovro Lugovic and Fabrizio Montesi. 2023. Real-world choreographic programming: An experience report. Retrieved from https://2303.03983

Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer. https://doi.org/10.1007/3-540-10235-3

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. Retrieved from http://www.fabriziomontesi.com/files/choreographic_programming.pdf

Fabrizio Montesi. 2015. Kickstarting choreographic programming. In *Proceedings of the 11th International Workshop on Web Services, Formal Methods, and Behavioral Types (WS-FM'14) and 12th International Workshop (WS-FM/BEAT'15) (Lecture Notes in Computer Science)*, Thomas T. Hildebrandt, António Ravara, Jan Martijn E. M. van der Werf, and Matthias Weidlich (Eds.), Vol. 9421. Springer, 3–10. https://doi.org/10.1007/978-3-319-33612-1_1

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

Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2014. Service-oriented programming with jolie. In *Web Services Foundations*, Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel (Eds.). Springer, 81–107. https://doi.org/10.1007/978-1-4614-7518-7_4

Fabrizio Montesi and Marco Peressotti. 2017. Choreographies meet communication failures. Retrieved from http://arxiv.org/abs/1712.05465.

Fabrizio Montesi and Nobuko Yoshida. 2013. Compositional choreographies. In *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR'13) (Lecture Notes in Computer Science)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.), Vol. 8052. Springer, 425–439. https://doi.org/10.1007/978-3-642-40184-8_30

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a higher kind. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*, Gail E. Harris (Ed.). ACM, 423–438. https://doi.org/10.1145/1449764.1449798

Tom Murphy VII, Karl Crary, and Robert Harper. 2007. Type-safe distributed programming with ML5. In *Proceedings of the 3rd Symposium on Trustworthy Global Computing (TGC'07) (Lecture Notes in Computer Science)*, Gilles Barthe and Cédric Fournet (Eds.), Vol. 4912. Springer, 108–123. https://doi.org/10.1007/978-3-540-78663-4_9

Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. 2004. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS'04)*. IEEE Computer Society, 286–295. https://doi.org/10.1109/LICS.2004.1319623

James Murty. 2008. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media.

Maurice Naftalin and Philip Wadler. 2006. *Java Generics and Collections*. O'Reilly Media.

Matthias Neubauer and Peter Thiemann. 2005. From sequential programs to multi-tier applications by program transformation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, Jens Palsberg and Martín Abadi (Eds.). ACM, 221–232. https://doi.org/10.1145/1040305.1040324

Sam Newman. 2021. *Building Microservices*. O'Reilly Media.

Rumyana Neykova and Nobuko Yoshida. 2017. Let it recover: Multiparty protocol-induced recovery. In *Proceedings of the 26th International Conference on Compiler Construction*, Peng Wu and Sebastian Hack (Eds.). ACM, 98–108. Retrieved from http://dl.acm.org/citation.cfm?id=3033031

Object Management Group. 2011. Business Process Model and Notation. Retrieved from http://www.omg.org/spec/BPMN/2.0/

Peter W. O'Hearn. 2018. Experience developing and deploying concurrency analysis at Facebook. In *Proceedings of the 25th International Symposium on Static Analysis (SAS'18) (Lecture Notes in Computer Science)*, Andreas Podelski (Ed.), Vol. 11002. Springer, 56–70. https://doi.org/10.1007/978-3-319-99725-4_5

OpenID Foundation. 2014. OpenID Specification. Retrieved from https://openid.net/developers/specs/

OpenJDK. May 2020. Loom—Fibers, Continuations, and Tail-Calls for the JVM. Retrieved from https://openjdk.java.net/projects/loom/

Tomas Petricek and Jon Skeet. 2009. *Real World Functional Programming: With Examples in F# and C*. Manning Publications.

Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A verified, end-to-end compiler for a Choreographic language. In *Proceedings of the 13th International Conference on Interactive Theorem Proving (ITP'22) (LIPIcs)*, June Andronick and Leonardo de Moura (Eds.), Vol. 237. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 27:1–27:18. https://doi.org/10.4230/LIPIcs.ITP.2022.27

Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. 2007. Towards the theoretical foundation of choreography. In *Proceedings of the World Wide Web Conference (WWW'07)*. IEEE Computer Society Press, US, 973–982.

Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A linear decomposition of multiparty sessions for safe distributed programming. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP'17) (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 502–516. https://doi.org/10.1145/3314221.3322484

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: A language for programming the web 2.0. In *Proceeedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, Peri L. Tarr and William R. Cook (Eds.). ACM, 975–985. https://doi.org/10.1145/1176617.1176756

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic programming for all (functional pearl). Retrieved from https://arXiv:2303.00924

Manu Sporny, Toby Inkster, Henry Story, Bruno Harbulot, and Reto Bachmann-Gmür. 2011. Webid 1.0: Web identification and discovery. W3C Editors Draft. https://www.w3.org/2005/Incubator/webid/spec/drafts/ED-webid-20111212/

Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. 2022. Global type inference for featherweight generic Java. In *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP'22) (LIPIcs)*, Karim Ali and Jan Vitek (Eds.), Vol. 222. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 28:1–28:27. https://doi.org/10.4230/LIPIcs.ECOOP.2022.28

K. Narendra Swaroop, Kavitha Chandu, Ramesh Gorrepotu, and Subimal Deb. 2019. A health monitoring system for vital signs using IoT. *Internet Things* 5 (2019), 116–129. https://doi.org/10.1016/j.iot.2019.01.004

Vasco T. Vasconcelos, Francisco Martins, Hugo-Andrés López, and Nobuko Yoshida. 2022. A type discipline for message passing parallel programs. *ACM Trans. Program. Lang. Syst.* 44, 4 (2022), 26:1–26:55. https://doi.org/10.1145/3552519

A. Laura Voinea, Ornela Dardha, and Simon J. Gay. 2020. Typechecking Java protocols with [St]Mungo. In *Proceedings of the 40th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'20), held as part of the 15th International Federated Conference on Distributed Computing Techniques (DisCoTec'20) (Lecture Notes in Computer Science)*, Alexey Gotsman and Ana Sokolova (Eds.), Vol. 12136. Springer, 208–224. https://doi.org/10.1007/978-3-030-50086-3_12

W3C. 2004. WS Choreography Description Language. Retrieved from http://www.w3.org/TR/ws-cdl-10/

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 129:1–129:30. https://doi.org/10.1145/3276499

Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP'19) (LIPIcs)*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 3:1–3:29. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A survey of multitier programming. *ACM Comput. Surv.* 53, 4 (2020), 81:1–81:35. https://doi.org/10.1145/3397495

Bill Wilder. 2012. *Cloud Architecture Patterns: Using Microsoft Azure*. O'Reilly Media.

Derek Wyatt. 2013. *Akka Concurrency*. Artima Incorporation, Sunnyvale, CA.

Received 27 June 2022; revised 24 April 2023; accepted 20 October 2023