Synthesising Java Implementations of Choreographic Protocols

Anne Møller Madsen and Fabrizio Montesi

University of Southern Denmark
anmad17@student.sdu.dk fmontesi@imada.sdu.dk

1 Introduction and Related Work

One of the best practices for the development of microservices is to coordinate them by following *choreographies*: coordination plans that prescribe how processes in a distributed system should interact with each other, by exchanging messages [Dragoni et al., 2017]. However, writing programs that comply with a choreography falls under the shadow of writing correct concurrent and distributed software, which is notoriously hard even for experts [Leesatapornwongsa et al., 2016]. This is due to the well-known state explosion problem: even for small programs, the number of possible ways in which they could interact can grow exponentially and reach unmanageable numbers [Clarke et al., 2011, O'Hearn, 2018].

Choreographic programming is an emerging programming paradigm where programs are choreographies [Montesi, 2013]. Its aim is to relieve programmers from implementing choreographies manually: first, programmers can code the choreography that they wish for by using a programming language equipped with primitives that make interactions syntactically manifest; then, a compiler automatically generates a working implementation of the choreography. The theory of choreographic programming has been explored in several directions, including service-oriented computing [Carbone and Montesi, 2013], adaptability [Dalla Preda et al., 2017], cyber-physical systems López and Heussen [2017], functional correctness [Jongmans and van den Bos, 2022], and security [Lluch-Lafuente et al., 2015, Bruni et al., 2021].

Choreographic programming languages are inspired by security protocol notation (also known as Alice and Bob notation), which was introduced for the definition of security protocols by Needham and Schroeder [1978]. The key primitive of these languages is the interaction term

```
A.expr -> B.x
```

which reads "`A` communicates the result of expression `expr` to `B`, which stores it in its local variable `x`". The participants `A` and `B` are called processes, or roles.

Until recently, implementations of choreographic programming languages mainly generated standalone systems and did not provide means to integrate the output code with mainstream development practices [Montesi, 2013, Carbone and Montesi, 2013, Dalla Preda et al., 2017]. The Choral language was later proposed as the first choreographic programming language that can be applied to mainstream programming [Giallorenzo et al., 2020]. In Choral, a choreography is compiled to a Java library for each process described in the choreography. A developer can then import this library and invoke it to play the part of that process in a distributed system.

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

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

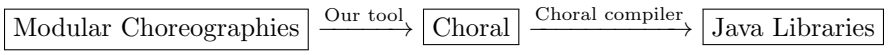
The previous line of code reads "variable `x` at `B` is assigned the result of invoking the static method `expr` of `MyClass` at `A` and passing it through an invocation of method `com` of object `channel` (which moves data from `A` to `B`)". Understanding Choral code thus requires more knowledge. While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

1.1 This presentation

We present work in progress on a new choreographic toolchain that aims at bridging the typical simplicity of choreographic programming with the practicality of Choral. Specifically, we will introduce the following contributions:

- A new choreographic programming language, called Modular Choreographies. Modular Choreographies offers a simple choreographic syntax with the standard "Alice and Bob" communication primitive (`A.expr -> B.x`), augmented with linguistic constructs for writing parametric procedures. The language is based on the choreographic programming theories of Procedural Choreographies [Cruz-Filipe and Montesi, 2017] and Recursive Choreographies [Montesi, 2022]. We extend these theories with linguistic constructs for using procedures more modularly, in particular the data and functions available at processes are locally scoped (in the previous theories, they are globally scoped). We also develop a type system (not presented in the remainder, but that will be shown in the presentation) which checks that procedures are invoked correctly: the processes that enact a procedure have access to the right data and local functions (e.g., for encryption).
- An implementation of Modular Choreographies, consisting of a parser and a type checker.
- A tool that, given code in Modular Choreographies, synthesises a program in Choral. The synthesiser automatically generates classes, methods, and the necessary usages of channels in order to move data correctly as instructed by the choreography.

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



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

Using Choral as intermediate technology also gives the pragmatic advantage of reusing what already exists to a reasonable extent. In particular, instead of adding yet another implementation of how choreographies can be distributed over separate programs for the participants—a process typically called Endpoint Projection in the literature of choreographies [Carbone et al.,

2012]. This allowed us to focus on the design of Modular Choreographies and the novel aspect of connecting "Alice and Bob" choreographies to object orientation.

In the rest of this paper (which consists of the next section), we give a taste of how our language and tool work by means of a running example based on a well-known security protocol. At the conference, we would like to get feedback on protocols that the community of microservices might find interesting to be written in Modular Choreographies, in order to create a future repository of reusable implementations.

2 Example

We exemplify how our toolchain for Modular Choreographies works with an example inspired by the Diffie-Hellman protocol for key exchange [Diffie and Hellman, 1976]. In particular, we adapt the choreography of this protocol written in [Montesi, 2022] to the syntax of Modular Choreographies.

In the protocol two participants (here processes), called Alice and Bob, establish a shared secret key over a public (insecure) channel. Alice and Bob compute the shared key by exchanging the results of computations based on secret data. Before execution the processes agree on two numbers: p and g, where p is a prime number, and g is a primitive root modulo p. Each process also have a secret number each, called a for Alice and b for Bob. The protocol then is as follows. Alice communicates the result of g^a modulo p to Bob, who stores it as x. Then, vice versa, Bob communicates to Alice the result of g^b modulo p. Finally, Alice and Bob can each compute the same shared key by using the number they got from the other process. For Bob that would be $s = x^b \bmod p$ [Montesi, 2022].

The protocol can be expressed in Modular Choreographies as follows.

Listing 1: Diffie-Hellman written in Modular Choreographies

```
1 DiffieHellman(
2     Alice:proc(modPow:(int,int,int) -> int, g:int, p:int, a:int),
3     Bob:proc(modPow:(int,int,int) -> int, g:int, p:int, b:int)
4 ):(Alice:proc(int), Bob:proc(int)) {
5     Alice.modPow(g,a,p) -> Bob.x;
6     Bob.modPow(g,b,p) -> Alice.y;
7     return Alice.modPow(y,a,p), Bob.modPow(x,b,p);
8 }
```

In Line 1, we start the definition of the choreographic procedure `DiffieHellman`, which has two parameters: a process `Alice` (denoted by the type `proc`), which must have access to a local function `modPow` (which takes three integers and returns an integer) and the three integers `g`, `p`, and `a` (Line 2); and a process `Bob`, with similar signature (Line 3). The procedure will return an integer at `Alice` and an integer at `Bob` (Line 4). Line 5 implements the communication from `Alice` to `Bob`, and vice versa in Line 6. Function `modPow` is supposed to compute modular exponentiation. In Line 7, the procedure returns the result of computing the shared secret key at `Alice` and `Bob`. Notice that the shared key computed by the two processes is the same, i.e., `modPow(y,a,p)` and `modPow(x,b,p)` return the same result [Needham and Schroeder, 1978].

Given the procedure in listing 1, our tool generates the code shows in listing 2.[1]. It is a Choral class with one method called `run`. In general, each procedure is compiled into a class,

[1]The Choral code that we show is the one output by Choral's pretty printer, modulo some modifications to whitespacing and parentheses for the sake of presentation.

and it is possible to have multiple procedures in the source program in Modular Choreographies (thus yielding more classes in output).

When making the compilation from Modular Choreographies to Choral the types are translated into Java types. An example is that the function `modPow` is translated to the type `Function3`, a Java interface that we provide which can be instantiated with a lambda expression that takes three arguments. Whenever possible, we reuse standard Java types, e.g., the primitive type `int` is translated to the class *Integer*. The functions and variables defined in each process in the Modular Choreography are added to the parameters of the `run` method in the generated Choral class. When functions and variables have the same name in different processes the name of the process is added to be able to distinguish between them, for example `g_Alice` on line 6 and `g_Bob` on line 9 in listing 2.

Communications in Choral must take place through appropriate channel objects that can move data from one process to another. We take the necessary channels as parameters of the `run` method, and the communication primitive `->` is then translated to invocations of the `com` method of the appropriate channel (where we also have to infer the type of data to be exchange, e.g., `Integer` in lines 11 and 12). The return instruction is translated into a tuple. This is because it is possible to return multiple values with Modular Choreographies. It is therefore needed to add a tuple class the size of the number of processes that returns a value. This makes it possible to return multiple values in Choral, mimicking the Modular Choreographies. Each process can also return multiple values, in this scenario the Choral method returns a tuple of tuples.

Listing 2: Result of the transformation from Modular Choreographies to Choral

```
1  public class DiffieHellman@(Alice, Bob) {
2      public static Tuple2@(Alice, Bob)<Integer, Integer>
3      run(
4          SymChannel@(Alice, Bob)<Integer> chAB,
5          Function3@Alice<Integer, Integer, Integer, Integer>
               modPow_Alice,
6          Integer@Alice g_Alice, Integer@Alice p_Alice,
7          Integer@Alice a,
8          Function3@Bob<Integer, Integer, Integer, Integer>
               modPow_Bob,
9          Integer@Bob g_Bob, Integer@ Bob p_Bob, Integer@Bob b
10     ) {
11         Integer@Bob x = chAB.<Integer>com(
12             modPow_Alice.apply(g_Alice, a, p_Alice)
13         );
14         Integer@Alice y = chAB.<Integer>com(
15             modPow_Bob.apply(g_Bob, b, p_Bob)
16         );
17         return new Tuple2@(Alice, Bob)<Integer,Integer>(
18             modPow_Alice.apply(y, a, p_Alice),
19             modPow_Bob.apply( x, b, p_Bob )
20         );
21       }
22 }
```

From the code in listing 2, the Choral compiler outputs two libraries, one for Alice and one for Bob. We show the interesting code for Alice in listing 3.

Listing 3: Result of the compilation from Choral to Java, for the process Alice

```
1  public class DiffieHellman_Alice {
2    // ...
3    public static Tuple2_R1 <Integer, Integer>
4    run(
5        SymChannel_A <Integer> chAB,
6        Function3 <Integer, Integer, Integer, Integer> modPow_Alice,
7        Integer g_Alice, Integer p_Alice, Integer a
8    ) {
9      chAB.<Integer>com(
10         modPow_Alice.apply(g_Alice, a, p_Alice)
11     );
12     Integer y;
13     y = chAB.<Integer>com(Unit.id);
14     return new Tuple2_R1<Integer, Integer>(modPow_Alice.apply(y, a,
           p_Alice), Unit.id);
15   }
16 }
```

The key idea is that each statement in the Choral source code, which involves both Alice and Bob, is compiled to Java code that plays the part that Alice needs to such that the choreographic statement is implemented. This can be seen, for example, by the fact that in line 9 the channel is used for sending (the data to be sent is provided), whereas in line 13 the channel is used for receiving. Parts that regard Bob are abstracted away as `Unit`s. In particular, the shared key that Bob computes is not known to Alice (albeit it is guaranteed to be the same by the protocol), and therefore in the returned tuple the second element is just a unit.

The classes compiled from Choral for Alice and Bob can then be used by developers to interact correctly according to the source choreography that we have originally written in Modular Choreographies (by invoking method `run`).

Acknowledgements

We thank Saverio Giallorenzo and Marco Peressotti for useful comments and discussions. Work partially supported by Villum Fonden, grant no. 29518.

References

A. Bruni, M. Carbone, R. Giustolisi, S. Mödersheim, and C. Schürmann. Security protocols as choreographies. In D. Dougherty, J. Meseguer, S. A. Mödersheim, and P. D. Rowe, editors, *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday*, volume 13066 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2021. doi: 10.1007/978-3-030-91631-2_5.

M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

E. M. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model checking and the state explosion problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi: 10.1007/978-3-642-35746-6_1.

L. Cruz-Filipe and F. Montesi. Procedural choreographic programming. In A. Bouajjani and A. Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017. doi: 10.1007/978-3-319-60225-7_7.

M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017.

W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22 (6):644–654, 1976.

N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

S. Giallorenzo, F. Montesi, and M. Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020.

S. Jongmans and P. van den Bos. A predicate transformer for choreographies—computing preconditions in choreographic programming. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Proceedings*, volume To appear of *Lecture Notes in Computer Science*. Springer, 2022.

T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016.

A. Lluch-Lafuente, F. Nielson, and H. R. Nielson. Discretionary information flow control for interaction-oriented specifications. In N. Martí-Oliet, P. C. Ölveczky, and C. L. Talcott, editors, *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi: 10.1007/978-3-319-23165-5_20.

H. A. López and K. Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In A. Seffah, B. Penzenstadler, C. Alves, and X. Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 437–443. ACM, 2017. doi: 10.1145/3019612.3019656.

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

F. Montesi. *Introduction to Choreographies*. Accepted for publication by Cambridge University Press, 2022.

R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978. ISSN 0001-0782. doi: 10.1145/359657.359659.

P. W. O'Hearn. Experience developing and deploying concurrency analysis at facebook. In A. Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2018.