Multiparty Languages: The Choreographic and Multitier Cases

Saverio Giallorenzo
Department of Computer Science and Engineering, Università di Bologna, Italy and INRIA, France

Fabrizio Montesi
Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Marco Peressotti
Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

David Richter
Technical University of Darmstadt, Germany

Guido Salvaneschi
University of St.Gallen, Switzerland

Pascal Weisenburger
Technical University of Darmstadt, Germany

Abstract

Choreographic languages aim to express multiparty communication protocols, by providing primitives that make interaction manifest. Multitier languages enable programming computation that spans across several tiers of a distributed system, by supporting primitives that allow computation to change the location of execution. Rooted into different theoretical underpinnings—respectively process calculi and lambda calculus—the two paradigms have been investigated independently by different research communities with little or no contact. As a result, the link between the two paradigms has remained hidden for long.

In this paper, we show that choreographic languages and multitier languages are surprisingly similar. We substantiate our claim by isolating the core abstractions that differentiate the two approaches and by providing algorithms that translate one into the other in a straightforward way. We believe that this work paves the way for joint research and cross-fertilisation among the two communities.

2012 ACM Subject Classification Computing methodologies → Distributed programming languages; Theory of computation → Distributed computing models; Software and its engineering → Multiparadigm languages; Software and its engineering → Concurrent programming languages; Software and its engineering → Distributed programming languages

Keywords and phrases Distributed Programming, Choreography Programming, Multitier Programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.23

Category Pearl

Funding *Fabrizio Montesi*: Villum Fonden, grant no. 29518, and Independent Research Fund Denmark, grant no. 0135-00219.
Marco Peressotti: Villum Fonden, grant no. 29518, and Independent Research Fund Denmark, grant no. 0135-00219.

1 Introduction

Programming concurrent and distributed systems is notoriously hard. Among other issues, it requires dealing with coordination and predicting how multiple participants will interact at

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:27
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

runtime, for which programmers do not receive adequate help from mainstream programming abstractions and technology [25, 21, 32].

The quest for finding elegant languages and methodologies that can help with concurrent and distributed programming has been a major focus of the research community for decades, including the seminal actor model and calculus of communicating systems [17, 27]. In this work, we are interested in two kinds of languages that have been recently gaining attention: *choreographic languages* [28, 2] and *multitier languages* [40]. Choreographic languages are designed to express multiparty communication protocols, by providing primitives that make interaction manifest. On the other hand, multitier languages allow for programming computation that spans across several tiers of a distributed system, by providing primitives that allow computation to change location of execution.

Both choreographic and multitier languages aim at making concurrent and distributed programming more effective, and have inspired several research and industrial language designs. However, choreographic and multitier languages stem from different ideas; they adopt different terminologies; they look different; they have evolved different features; and they have found different applications in practice. Perhaps because the design principles of choreographic and multitier languages come from different angles, the two communities have prolifically evolved independently. However, as a consequence, the commonalities and actual differences between the two research lines remain unclear, which impedes cross-fertilisation.

In this paper, we offer a new perspective on the relationship between choreographic and multitier languages. We show that, despite their different starting points and evolutions, they share a strong core idea that classifies them both as what we call *multiparty languages*—languages that describe the behaviour of multiple participants. Leveraging this commonality, it is possible to derive choreographic programs from multitier programs, and vice versa. Our aim is to provide a way for each community to access the other, encouraging cross-fertilisation.

We outline our investigation and contributions:

- In Section 2, we give an overview of the essential features of choreographic and multitier languages. We recap the history of the two approaches and identify their key differences, which lie in perspective (objective vs subjective) and in the modelling of communications (manifest vs non-manifest). We also pinpoint the commonality that classifies choreographic and multitier languages as multiparty.
- In Section 3, we present an example use case for both choreographic and multitier programming, which introduces the concrete choreographic and multitier programming languages that we will use in the rest of our development: Choral [16] and ScalaLoci [38].
- In Section 4, we introduce Mini Choral and Mini ScalaLoci, two representative but minimal languages for choreographic and multitier programming, respectively. Mini Choral and Mini ScalaLoci dispense with the features that are not essential parts of their respective paradigms, which allows us to study how the essential differences can be bridged in the next section.
- In Section 5, we define algorithms for translating programs in Mini Choral to programs in Mini ScalaLoci, and vice versa. The translations deal with the changes in perspective and manifestation of communications between the two paradigms. For example, translating a multitier program into a choreographic one requires synthesising a communication protocol that enacts the necessary communications among participants.

  Our translations are not just of inspiration to see the connection between the two paradigms (which we leverage in the next section), but also open a window towards the future sharing of theoretical and practical results. An example for each direction: by translating a multitier program into a choreographic one and then using a choreographic compiler to generate executable code, we can know statically the pattern of communications that will be enacted by the executable code (this property is called "Choreography Compliance" [16] or "EndPoint Projection Theorem" [3]); by translating a choreographic program into a multitier one and then using a multitier compiler to generated executable code, we can reuse all the machinery developed by the multitier community to generate code for different technologies (e.g., the code generated for one participant is in JavaScript for a web browser while the code for another might be code runnable on the Java Virtual Machine for a server).

- Our study shows that, while choreographic and multitier programming languages are different enough to be independently useful, they are also near enough to benefit from cross-fertilisation. In Section 6, we report on important features that have been developed separately in the choreographic and multitier research lines. We find that important features for the development of concurrent and distributed systems have been developed for one paradigm but not the other. Inspired by our newfound connection, we discuss how these features could be ported over to the other paradigm in the future, setting up future work enabled by our view.

2 Background: Choreographic and Multitier Programming Languages

In this section, we give some background on choreographic and multitier languages, and discuss their differences and similarities.

2.1 Choreographic Languages

Choreographic languages are inspired by the famous "Alice and Bob" notation, or security protocol notation [30]. The idea is to define how the different participants of a system should communicate (or interact)—which later inspired also message sequence charts and sequence diagrams [20]. Textual and graphical choreographic languages have already been adopted in industry as specification languages in different settings ranging from business processes, e.g., the choreographic language in OMG's Business Process Model and Notation, to web services, e.g., W3C's Web Services Choreography Description Language [31, 37].

The essence of a choreographic language is the capability of expressing explicitly data flows from a participant to another through communication, and of composing such communications into larger structures. In other words, choreographies make interaction and the structure of interaction protocols *manifest*. A communication from a participant, `Alice`, to another, `Bob`, is written as follows.

```
Alice.userId -> Bob.x : ch
```

The statement above reads: `Alice` sends its `userId` (a local variable storing a user identifier) to `Bob`, which stores it in its local variable `x`, and the communication takes place through the channel `ch`.

Communication statements can be composed in larger and more sophisticated protocols, for example using the sequential operator "`;`". In the following protocol snippet: after interacting with `Alice`, `Bob` forwards to `Charlie` the user identifier that it received through a separate channel `ch2`.

Listing 1 A simple choreography with three participants.

```
Alice.userId -> Bob.x : ch;
Bob.x -> Charlie.y : ch2;
```

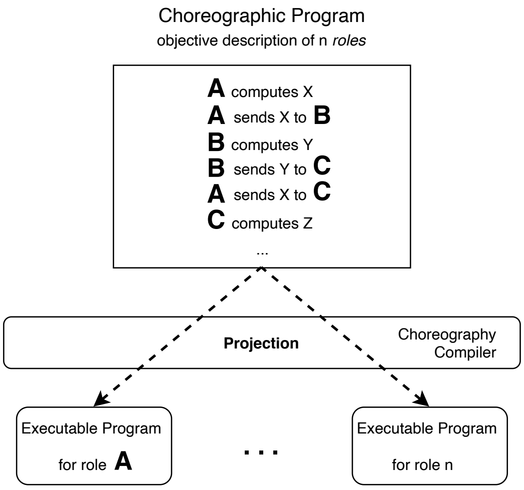


Figure 1 Choreographic Programming.

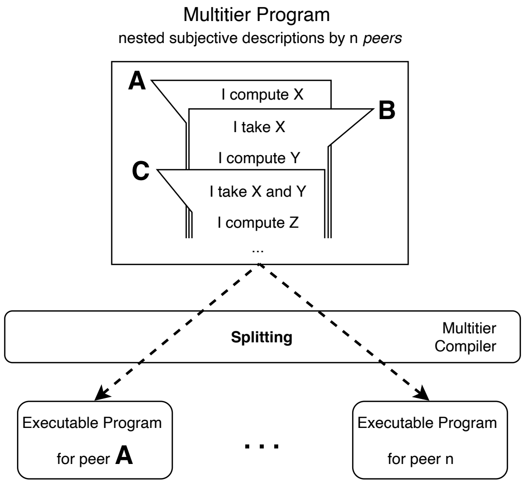


Figure 2 Multitier Programming.

In the paradigm of *choreographic programming* [28], choreographic languages are full-fledged programming languages: developers write the implementation of an entire multiparty system as a choreography, and then a compiler automatically generates an executable program for each participant. This process is depicted in Figure 1. Choreographies resemble play scripts, written from an external point of view, describing the interactions among all participants. We call this view *objective*. Participants, like `Alice` and `Bob`, are typically referred to as *roles* in choreographies, and the procedure that generates the executable program for each role is called projection (or endpoint projection) [4, 11].

The code in Listing 1 is valid code in the Chor language, the first implementation of choreographic programming [28, 4]. Chor targets microservices: given that code (with appropriate boilerplate), Chor would generate executable programs of microservices that implement `Alice`, `Bob`, and `Charlie`. Choreographic programming has been applied to other settings, e.g., information flow [22], parallel algorithms [10], cyber-physical systems [24, 23], runtime adaptation [11], and integration processes [15].

2.2 Multitier Languages

Multitier languages are inspired by one of the ideas proposed with ambient calculi [5]. In this kind of process calculi, terms express the place (the "ambient") at which computation occurs. Computations that take place at different locations can be nested, which enables describing multiparty systems. It was later shown that the idea can be combined with well-known abstractions, by developing a variation of λ-calculus with locations called Lambda 5 [29]. This solution prompted the development of *multitier languages* [36, 8, 40], which extend existing programming languages with locations. The term multitier comes from the fact that these languages were mostly developed for web programming, where tiers is used to refer to the typical participants of a web system (e.g., client, backend server, and database).

The crux of a multitier language is the capability of hopping from the point of view of a participant to that of another—the multitier language by Serrano et al. is aptly called "Hop" [36]. When hopping from a participant to another, it is possible to move data from the participant that we are leaving to the participant that we are going to—enabling communication. As an example, consider a remote procedure call from a client to server. In a recent incarnation of multitier programming that builds on the Scala language, ScalaLoci [38], this can be written as follows.[1]

```
def rpc(input: String): String on Client = on[Client] {
  val result =
    on[Server].run.capture(input) {
      expensiveFunction(input)
    }.asLocal
  return result
}
```

Participants are referred to as peer types in ScalaLoci. The method `rpc` above is defined as a block of code that starts at the client peer (`on[Client]`). The client stores the result of some computation in its local variable `result`, but this computation is performed at the server. This result is achieved by "moving" to the server with the instruction `on[Server]`. The invocation of method `run`, right afterwards, models some computation, and `capture(input)` means that we want to move the content of the local variable `input` from the peer that we are leaving (the client) to the one that we are going to (the server). How this move is achieved is left to the implementation (ScalaLoci generates a communication strategy automatically). The server then runs an expensive function on the input, and the execution goes back to the client—the code block at the server ends. The invocation of `asLocal` ensures that the return value of the code at the server is moved to the location of the enclosing scope (the client). We finally return the result at the client.

Like choreographic programming languages, multitier languages come with a compiler that turns the multiparty view of the system into executable programs. This process is depicted in Figure 2. Given a multitier program, a multitier compiler generates an executable program for each peer type (in the case of Section 2.2, these would be client and server). The procedure for generating code is called splitting. The nested "dialogues" of peers inside the multitier program depict that a multitier program has many viewpoints, switching regularly from the point of view of a peer to that of another. Nevertheless, code is written with the viewpoint of the peer we are currently in. For this reason, we say that multitier programs adopt a nested *subjective* view.

2.3 Towards Linking Choreographic to Multitier Languages

The two communities of choreographic and multitier languages have prolifically evolved independently [2, 40]. They adopted different design principles, and they have found different practical applications—most notably service-oriented computing for choreographies and web development for multitier programming. As a result, they have also developed several features independently (we discuss some of the most important ones in Section 6). In addition, the two communities have been facing different challenges. For example, multitier programming languages historically tackle the problem of "impedance mismatch": the necessity of handling data conversions and heterogeneous execution engines in the web (the Google Web Toolkit is a multitier framework that contributes to this research area). Instead, choreographic programming mainly aimed at achieving "choreography compliance": providing the guarantee that distributed systems communicate as expected and with desirable properties (like liveness).

Yet, the two paradigms are clearly linked. We drew Figure 1 and Figure 2 with the intention of highlighting such connection. Indeed, despite differences in both terminologies

[1] For simplicity of presentation, we omit library calls that would be necessary to deal with asynchrony.

and methods, the strategies of choreographic and multitier programming languages share a similarity: both define the behaviour of a multiparty system in a single compilation unit, and then offer ways to synthetise executable implementations for the participants. We thus identify both kinds of languages as instances of the larger class of *multiparty languages*—leaving the class open to future additions. We see value in both techniques for multiparty programming. In choreographies protocols are manifest, which makes them easy to understand. Multitier programs give access to multiparty programming with a developing experience that resembles standard "local programming" by leveraging scoping.

Despite both choreographic and multitier languages sharing the multiparty approach, they remain pretty diverse in terms of theoretical background. The theory of choreographic language typically stands on process calculi, whereas multitier models build on λ-calculus [18, 4, 19, 11, 40]. This is likely an important reason why the link between choreographic and multitier languages has been overlooked for long. Very recently, however, it has been shown that object-oriented languages can be extended to capture choreographies, by generalising the notion of data type to data types located at *multiple roles* [16]. In the resulting language, called Choral, a choreography among a few roles can be expressed as an object. For example, we can write the choreography in Listing 1 in Choral as follows:

```
class Example@(Alice, Bob, Charlie) {                // the three roles of the protocol
  DiDataChannel@(Alice,Bob)<Serializable> ch;        // channel from Alice to Bob
  DiDataChannel@(Bob,Charlie)<Serializable> ch2;     // channel from Bob to Charlie

  /* constructor omitted */

  public UserID@Charlie run(UserID@Alice userId) {  // the protocol
    UserID@Bob x = ch.<UserID>com(userId);           // Alice.userId -> Bob.x : ch
    return ch2.<UserID>com(x);                       // Bob.x -> Charlie.y : ch2
  }
}
```

Briefly—as we give a more detailed description of Choral programs in Section 3.2—the `Example` class declares three roles (`Alice`, `Bob`, and `Charlie`) and two directed channels (`ch` from `Alice` to `Bob` and `ch2` from `Bob` to `Charlie`). These correspond to the roles and channels assumed in Listing 1. The protocol described in Listing 1 is implemented by method `run` that takes an instance of `UserID` located at `Alice` and returns one located at `Charlie` passing through `Bob`. Communication happens by invoking method `com` of the two channels.

Choral helps in leveling the playfield with multitier programming. Indeed, we now have an object-oriented incarnation of choreographic programming that we can use to compare to object-oriented multitier languages, here represented by ScalaLoci. In the next sections, we leverage this common ground and take Choral and ScalaLoci as representative languages for their respective paradigms.

3 Overview of Choral and ScalaLoci

In this section, we give an overview of the representative languages for choreographic and multitier programming that we have chosen, Choral and ScalaLoci, by using them to deal with a simple yet comprehensive example of a context-aware protocol for e-mail fetching.

3.1 A Context-Aware Email-Fetching Protocol

Before delving into the details of the two implementations, we discuss briefly the protocol that we want to program. A depiction as a sequence diagram is given in Figure 3. The protocol defines an interaction between an Email Client and an Email Server. Specifically,

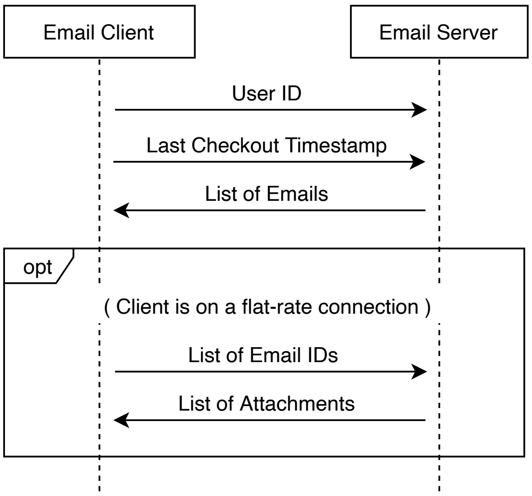


Figure 3 Sequence diagram for context-aware e-mail fetching.

the Client sends its identification token—here simplified as User ID—and the timestamp of the last e-mail checkout to the Server. The Server returns the list of e-mails received after the timestamp to the Client. After the above interaction, the Client and the Server enter an optional block. The optional block is executed depending on the context of the client, namely, if the connection from the Client to the Server is flat-rate, i.e., if the connection fee paid by the Client is independent from its usage. If that is the case, the Client sends the Server the list of e-mail IDs retrieved in the previous interaction to fetch their attachments. The Server concludes the optional part of the protocol by sending to the Client the requested attachments.

3.2 A Choreographic Programming Implementation with Choral

In Listing 2, we use Choral to implement the protocol from Figure 3. The example illustrates the main concepts of the choreographic programming approach and how Choral captures them in the object-oriented setting.

In Choral, objects have types of the form `T@(R1, ..., Rn)`, where `T` is the interface of the object (as usual), and `R1`, ..., `Rn` are the roles that collaboratively implement the object. As we see below, Choral supports two notations for object ownership: the standard form `@(A, ..., Z)` and the contracted form `@A`, for objects that belong to one role (shortcut for `@(A)`). Incorporating roles in data types makes distribution manifest at the type level.

In Listing 2, at Line 3, we define a class `EmailSystem` implemented by two roles: the `Client` and the `Server`. The method `updateEmails` (Line 8) implements the actual protocol from Figure 3. Lines 4–6 declare class-level `private` objects, i.e., accessible from the `updateEmails` method and other (omitted) ones within the class. Specifically, at Line 4, we have the `MailServerDB` located at the `Server`. At Line 5, we find the complementary `MailDB` of the `Client`. At Line 6, we define the object used to transfer data between the two roles: a `SymChannel`—standing for *symmetric channel*—shared between the two roles and able to transmit `Serializable` objects. We omit the initialisation of the abovementioned objects.

Considering the description of the implementation of the e-mail fetching protocol, we look at the `updateEmails` method (Line 8). The method does not return a value (`void`) and takes as input the `UserId`—which simplifies the user authentication procedure here, for brevity—to identify the user of the `Client` at the `Server`.

In the body, at Line 9, we pass the `UserId` to the `Server`. We do this by invoking the method `com` of the `ch SymChannel` giving to it as argument the `userId`. This is done by the expression `userId >> ch::com` which uses the Choral chaining operator `>>` and that corresponds to the expanded expression `ch.com(userId)`.[2] The method `com` of the `SymChannel` transfers the value of the sender given as input into an equivalent representation of the value at the receiver. In this case, the sender is the `Client` (where the `UserId` object lives) and the receiver is the `Server`, which stores the result of the communication into variable `id` which is an object of type `UserId` at its location—i.e., `UserId@Server`.

The transfer of the `Timestamp` from the `Client` to the `Server` is similar (Line 10): we retrieve the object from the `clientDB`—invoking method `lastCheckOut`—and we transfer it to the `Server` thought the `SymChannel`. Then, to fetch the e-mails, the `Client` receives a transmission from the `Server`. The `Server` interrogates its local database (`serverDB`) by extracting all e-mails belonging to the `id` of the `Client` and received `since` its last checkout (indicated by the `timestamp`) and sends them to the `Client` via their shared `SymChannel`. At Line 12, the `Client` uses the received list of `emails` to `update` its local database (`clientDB`).

Lines 13–20 implement the optional part of the protocol from Figure 3. First, the `Client` checks whether it is using a flat-rate connection—this is done through the static library `ClientLib` and its method `isOnFlatRate`.

The `if-else` block at Lines 13–20 allows us to explain the concept of knowledge of choice (a hallmark element of choreographic programming) and how Choral implements it. Briefly, the concept of knowledge of choice indicates a fork in the flow of a program among alternative behaviours, where the concerned roles should coordinate to ensure that they agree on which behaviour they should enact. In Choral, we adopt a standard choreographic solution to this problem [11] by defining a "selection" primitive to communicate constants drawn from a dedicated set of "labels", so that the compiler has enough information to build code that can react to choices made by other roles. Concretely, to define selections, Choral uses a method-level annotation `@SelectionMethod`[3], which developers can apply only to methods that can transmit instances of enumerated types between roles (the compiler checks for this condition). Conveniently, the `SymChannel` used in the example also supports selections via its `select` methods. In Listing 2, we find the implementation of the knowledge of choice of the conditional at Line 14 (where the `Client` "decides" to fetch the attachments) and at Line 20 (which skips the retrieval). In the example, we implement the choice by defining the `Choice` `enum` class at Line 1—note that we use the identifier `Role` for the single role that owns the `Choice` object in its declaration, instantiated at the `Client` at Lines 14 and 20.

If the `Client` uses a flat-rate connection, the chained statement at Lines 15–17 execute: first (Line 15) the `Client` sends to the `Server` the IDs of the e-mails (retrieved through `extractIds(emails)`) whose attachments it wants to retrieve, then (Line 16) the `Server` uses the received `ids` to extract from its database (`serverDB`) the attachments and it send them back to the `Client`, and finally (Line 17) the `Client` uses the received attachments to update its local database.

---

[2] To make Choral programs closer to standard choreographic notation, where data flows from left to right, Choral borrows the forward chaining operator `>>` from F#: `exp >> obj::method` is syntactic sugar for `obj.method(exp)`.

[3] Choral preserves the standard `@`-notation for annotations from Java, which is contextually separated from `@(R1,...,Rn)` parameters in Choral programs.

Listing 2 Choral implementation for the context-aware e-mail fetching example.

```
enum Choice@Role { THEN, ELSE }

class EmailSystem@(Client, Server) {
  private MailServerDB@Server serverDB = ...;
  private MailDB@Client clientDB = ...;
  private SymChannel@(Client, Server)<Serializable> ch = ...;

  void updateEmails(UserId@Client userId) {
    UserId@Server id = userId >> ch::com;
    Timestamp@Server timestamp = clientDB.lastCheckOut() >> ch::com;
    List@Client<Email> emails = serverDB.since(id, timestamp) >> ch::com;
    clientDB.update(emails);
    if (ClientLib@Client.isOnFlatRate()) {
      Choice@Client.THEN >> ch::select;
      clientDB.extractIds(emails) >> ch::com
        >> serverDB::getAttachments >> ch::com
        >> clientDB::updateAttachments;
    }
    else {
      Choice@Client.ELSE >> ch::select;
    }
  }
}
```

3.3 A Multitier Programming Implementation with ScalaLoci

We now use ScalaLoci to illustrate the multitier programming approach, implementing the protocol from Figure 3 in Listing 3.

In ScalaLoci, the location of different values is specified through *placement types*. The placement type `T on P` represents a value of type `T` on a peer `P`. Developers can freely define the different components, called *peers*, of the distributed system. For instance, in the example, `serverDB` is a `MailServerDB` placed on the `Server` (Line 5) and `clientDB` is a `MailDB` placed on the `Client` (Line 6).

Peers are defined as abstract type members (Lines 2 and 3). Further, peer types express the architectural relation between the different peers by specifying ties between peers, thus supporting generic distributed architectures. Ties statically approximate the run time connections between peers. In the example, we define a *single* tie from client to server (Line 2) and from server to client (Line 3). A single tie expresses the expectation that a single remote instance is always accessible. In the specified architecture, a client connects to a single server and a server program instance handles a single client.

The `updateEmails` method (Line 8) encapsulates the communication logic from Figure 3. It takes the `UserId` for identifying the client as input. The implementation diverts control flow to the server using a nested `on[Server].run` expression (Line 10). The `capture` clause transfers both the `timestamp` and the `userId` from the client to the server. Inside the server expression (Line 11), the server queries its local `serverDB` database to extract all e-mails belonging to the `userId` of the client received `since` its last checkout (indicated by the `timestamp`). The result of the server-side expression is returned to the client using `asLocal` (Line 12).

In ScalaLoci, accessing remote values via the `asLocal` marker creates a local representation of the remote value by transmitting it over the network. Usually, such local representation uses a future, accounting for network delay and potential communication failure. Futures represent values that become available in the future or produce an error. In the example, however, we picked a different transmission scheme where values are transmitted synchronously. ScalaLoci

Listing 3 ScalaLoci implementation for the context-aware e-mail fetching example.

```
1  @multitier object EmailSystem {
2    @peer type Client <: { type Tie <: Single[Server] }
3    @peer type Server <: { type Tie <: Single[Client] }
4
5    private val serverDB: MailServerDB on Server = ...
6    private val clientDB: MailDB on Client = ...
7
8    def updateEmails(userId: UserId): Unit on Client = on[Client] {
9      val timestamp: Timestamp = clientDB.latestCheckout
10     val emails: List[Email] = on[Server].run.capture(userId, timestamp) {
11       serverDB.since(userId, timestamp)
12     }.asLocal
13
14     clientDB.update(emails)
15     if (ClientLib.isOnFlatRate) {
16       val ids = clientDB.extractIds(emails)
17       clientDB.updateAttachments(
18         on[Server].run.capture(ids) { serverDB.getAttachments(ids) }.asLocal)
19     }
20   }
21 }
```

allows developers to choose among different such *transmitters*.

The client then uses the received list of `emails` to `update` its local `clientDB` database (Line 14). Lines 15–19 implement the optional part of the communication logic from Figure 3. If the client is currently using a flat-rate connection—as indicated by the static `ClientLib.isOnFlatRate` method—the client initiates a second server-side computations using `on[Server].run` (Line 18). The client transfers the IDs of the e-mails (retrieved through `extractIds(emails)`)—whose attachments to receive—to the server, which extracts the attachments from its `serverDB` database and returns them to the client, which then updates its local `clientDB` with the received attachments (Line 17).

4 Mini Choreographic and Multitier Languages

We now introduce Mini Choral and Mini ScalaLoci, minimal languages that omit most features of their reference counterparts that are irrelevant to our study (like generics and inheritance). This allows us to focus on the distinctive traits that characterise the choreographic and multitier approaches, respectively. The minimality of the two languages is instrumental to highlight their distinguishing features here and to focus on the salient points that define their reciprocal translations in Section 5. Next, we present the grammar of the two languages and briefly describe the components that mark them respectively as choreographic and multitier languages.

4.1 Mini Choral

Listing 4 displays the grammar of Mini Choral. C ranges over class declarations, *Channel* ranges over channel declarations, *Field* ranges over class fields, *Method* ranges over method definitions, *Type* ranges over type expressions, and *Exp* ranges over expression terms. The metavariable *id* ranges over both class names, fields, and variables. We use `A`, `B`, `C` to range over role names. Here and in the reminder of the paper, we use $\overline{\textit{overlines}}$ to denote sequences of terms of the same sort and we denote concatenation of sequences using a comma.

Listing 4 Syntax of Mini Choral

Mini Choral	C	$::=$	**`class`** id`@`$(\overline{\texttt{A}})\{\ \overline{Channel}\ \overline{Field}\ \overline{Method}\ \}$
Type Expression	$Type$	$::=$	id`@`$(\texttt{A})$
Channels	$Channel$	$::=$	`DiChannel@(A, B) ch_A_B`
Field	$Field$	$::=$	$Type\ id$
Method Definition	$Method$	$::=$	$Type\ id(\overline{Type\ id})\{$ **`return`** $Exp\ \}$
Expression	Exp	$::=$	$id \mid Exp.id \mid Exp.id(\overline{Exp}) \mid$ **`new`** id`@`$(\texttt{A})(\overline{Exp})$
		$\mid$	lit`@`$(\texttt{A}) \mid$ **`if`** $(Exp)\ \{\ Exp\ \}$ **`else`** $\{Exp\} \mid Exp;\ Exp$
		$\mid$	$ch\_A\_B.\texttt{com}(Exp) \mid ch\_A\_B.\texttt{select}(Exp)$

The class declaration C defines its name id, its owner roles $\overline{\texttt{A}}$ within the `@`$(\ \cdots)$ clause, the topology of directed channels available between roles in $\overline{Channel}$, its field declarations $\overline{Field}$, and its suite of method definitions $\overline{Method}$.

In Mini Choral, we decided to focus on describing *data flow* and to limit Choral's expressivity regarding *data distribution*. That is, we allow only the declared **`class`** to be distributed at multiple roles, while variables belong to only one role, with the exception of *Channel*s, which specify the network topology as a set of objects located (and able to transfer single-role objects) between two roles. Specifically, Mini Choral supports only one-way channels (drawn from Choral and called `DiChannels`) of the shape `DiChannel@(A,B) ch_A_B`—with `A` and `B` roles of the enclosing class. In this work, the loss of expressiveness of the Mini variant with respect to Choral—which supports the definition of multi-role classes/fields without the above limitations—lends itself to simplify the algorithms in our translation in Section 5 without losing generality on the choreographic approach. In the general case, Choral can express arbitrary channel topologies and user-defined implementations of communications semantics (e.g., asymmetric channels or bidirectional symmetric channels) [16]—whereas most choreographic languages assume a complete topology of channels between all roles in a choreography with a fixed communication semantics [4, 11].

Following the considerations above, we restrict type expressions *Type* to define variables located at one role id`@`$(\texttt{A})$. This is reflected in the definition of *Field*s but also in method definitions, where we additionally assume the return type *Type* and the types of arguments $\overline{Type\ id}$ to be located at the same role. The body of the method is the single statement **`return`** Exp. Regarding expressions, we focus our description on the relevant, non-standard elements: object creation **`new`** id`@`$(\texttt{A})(\overline{Exp})$ happens for classes at only one role and literals lit`@`$(\texttt{A})$ (integers, strings, etc.) are always located at one role. In Exp, we use $Exp; Exp$ to represent a block which evaluates the expression on the left, discards its value, and returns the evaluation of the expression on the right.

Although already captured by the grammar, we include channel invocations of the shape $ch\_A\_B.\texttt{com}(Exp)$ and $ch\_A\_B.\texttt{select}(Exp)$ to highlight their relevance in the language. `DirectChannels` support both methods `com`, meant to transfer data between two roles, and `select`, used to solve knowledge-of-choice challenges in conditionals (that is, informing a role of a local choice made by another role, e.g., by using a conditional) [16]. When using `selects`, we assume that the compiler provides us with a `Choice` **`enum`** class at one role, with a `THEN` and `ELSE` inhabitants (as presented at Line 1 in Listing 2).

4.1.1 Example: Mini Choral Expressiveness

We conclude the presentation of our minimal choreographic language by illustrating its expressiveness with respect to its reference Choral language with an implementation of the

Listing 5 Mini Choral implementation for the context-aware email fetching example.

```
class EmailSystem@(Client, Server) {
  DirectChannel@(Client, Server) ch_Client_Server
  DirectChannel@(Server, Client) ch_Server_Client

  MailServerDB@Server serverDB
  MailDB@Client clientDB

  Unit@Client updateEmails(UserId@Client userId) {
    return contextAwareUpdate(getEmails(userID, clientDB.lastCheckOut()))
  }

  List@Client getEmails(UserId@Client id, Timestamp@Client ts) {
    return ch_Server_Client.com(
      serverDB.since(ch_Client_Server.com(id), ch_Client_Server.com(ts))
  }

  Unit@Client contextAwareUpdate(List@Client emails) {
    clientDB.update(emails);
      if (ClientLib.isOnFlatRate()) {
        ch_Client_Server.select(Choice@Client.THEN);
        clientDB.updateAttachments(
          ch_Server_Client.com(
            serverDB.getAttachments(
              ch_Client_Server.com(clientDB.extractIds(emails)))))
      }
      else {
        ch_Client_Server.select(Choice@Client.ELSE); Unit
      }
  }
}
```

email-fetching protocol presented in Section 3.2, Listing 2.

We report the code of the Mini Choral implementation of the protocol in Figure 3 in Listing 5. In the Listing, the main notable difference with Listing 2 is that, by removing assignments, we rely on method bindings to reuse variables in "subsequent" (;) invocations. Although divided into three sub-methods, we find the `updateEmails` method that invokes the `getEmails` method, which fetches the emails from the `Server` by sending to it the `id` of the user and the timestamp (`ts`) of the last checkout and transmitting back the result of the extraction on the `serverDB`. Notice that the return type of the `getEmails` method omits the definition of the "content" of the list due to the lack of generics. As expected, by omitting generics we also drop support for specifying/checking the correct/expected content of the collection—an orthogonal guarantee with respect to the specification/check of the flow of data among roles. The lack of generics does not hamper the expressiveness of the language to capture the correct movement of the data from the `Server` to the `Client` and vice versa. After obtaining the emails, we can apply method `contextAwareUpdate` which updates the email database of the client and proceeds to conditionally retrieve the attachments of the fetched emails. This is done by informing the `Server` of the choice, via the `select` methods.

4.2 Mini ScalaLoci

Listing 6 displays the grammar of Mini ScalaLoci. *L* ranges over object declarations, *Peer* ranges over peer declarations, *Field* ranges over class fields, *Method* ranges over method definitions, *Type* ranges over type expressions, *PlacedType* ranges over placement type expressions, *Exp* ranges over expressions, and *PlacedExp* ranges over placed expressions. The

Listing 6 Syntax of Mini ScalaLoci

Mini ScalaLoci	L	::=	`@multitier object` { $\overline{Peer}\ \overline{Field}\ \overline{Method}$ }
Peer	$Peer$	::=	`@peer type` A <: { `type Tie` <: `Any` $\overline{\texttt{with Single[B]}}$ }
Placement Type Expression	$PlacedType$	::=	$Type$ `on` A
Type Expression	$Type$	::=	id
Field	$Field$	::=	`val` $id : PlacedType$
Method Definition	$Method$	::=	`def` $id\ (\overline{id : Type}) : PlacedType\ =\ PlacedExp$
Placed Expression	$PlacedExp$	::=	`on`$[A]$ { Exp }
Expression	Exp	::=	$id \mid Exp.id \mid Exp.id(\overline{Exp}) \mid$ `new` $id(\overline{Exp})$
		|	$lit \mid$ `if` (Exp) { Exp } `else` { Exp } $\mid Exp;\ Exp$
		|	`on`$[A]$.`run`.`capture`$(\overline{id})$ { Exp }.`asLocal`

metavariable id ranges over both class names, fields, and variables. We use `A`, `B`, `C` to range over peers.

The object declaration L defines its name id, and its peers $\overline{A}$ and topology of directed ties between the peers within the `@peer type` A `<: { type Tie <: Any` $\overline{\texttt{with Single}[A]}$ `}` clauses, its field declarations $\overline{Field}$, and its method definitions $\overline{Method}$. Fields associate a placement type expression $PlacedType$ to a variable.

Mini ScalaLoci is able to express different *topologies* rather than being restricted to *a fixed client-server* model. This choice remarks the departure taken by ScalaLoci from other multitier models and implementations [8, 9, 34, 35, 36], which assume a fixed client-server or n-tier architecture of an application. Contrarily, in ScalaLoci, the developer defines an arbitrary number of peers and directional ties between them. In contrast to ScalaLoci, Mini ScalaLoci only supports a single connected peer instance per peer type (drawn from ScalaLoci's `Single` ties) of the shape `@peer type A <: { type Tie <: Single[A] }`—with `A` and `B` peers of the enclosing multitier module. In this work, the loss of expressiveness of the Mini variant with respect to ScalaLoci lends itself to simplify the algorithms in our translation in Section 5 with losing generality on the multitier approach.

In method definitions, the return type $PlacedType$ specifies a location, which places the computation of the whole method on that peer, whereas the arguments only have types but no placement $\overline{id :\ Type}$. The body of the method is a placed expression $PlacedExp$ that specifies the placement of the contained expression Exp. Regarding expressions, we focus our description on the main differences with Choral: In ScalaLoci, we locate expressions rather than data and therefore neither instantiation `new` $id(\overline{Exp})$ nor literals lit (integers, strings, etc.) carry placement annotations.

Nested remote blocks are encoded by `on`$[A]$.`run`.`capture`$(\overline{id})$ { Exp }.`asLocal` expressions, which execute the nested expression on the peer A and returns its result via `asLocal` to the surrounding peer, i.e., switching the current perspective to another peer for evaluating the nested expression. Note that in the Mini variant, we keep the `run`, `capture` and `asLocal` constructs to be close to the complete version of the ScalaLoci language (that is syntactically more flexible and supports optional `capture` clauses and `asLocal` on module-level value bindings).

4.2.1 Example: Mini ScalaLoci Expressiveness

We show the implementation of the email-fetching example presented in Section 3.3, Listing 3 using our minimal multitier language to demonstrate its expressiveness with respect to its reference ScalaLoci language.

Listing 5 shows the Mini ScalaLoci implementation of the communication scheme in

Listing 7 Mini ScalaLoci implementation for the context-aware email fetching example.

```
@multitier object EmailSystem {
  @peer type Client <: { type Tie <: Any with Single[Server] }
  @peer type Server <: { type Tie <: Any with Single[Client] }

  val serverDB: MailServerDB on Server
  val clientDB: MailDB on Client

  def updateEmails(userId: UserId): Unit on Client = on[Client] {
    contextAwareUpdate(getEmails(userID, clientDB.lastCheckOut()))
  }

  def getEmails(id: UserId, ts: Timestamp): List on Client = on[Client] {
    on[Server].run.capture(id, ts) {
      serverDB.since(
        on[Client].run.capture(id) { id }.asLocal,
        on[Client].run.capture(ts) { ts }.asLocal))
    }.asLocal
  }

  def contextAwareUpdate(emails: List): Unit on Client = on[Client] {
    clientDB.update(emails);
    if (ClientLib.isOnFlatRate()) {
      clientDB.updateAttachments(on[Server].run.capture(emails) {
        serverDB.getAttachments(
          on[Client].run.capture(emails) { clientDB.extractIds(emails) }.asLocal)
      }.asLocal)
    }
    else {
      Unit
    }
  }
}
```

Figure 3. As with Mini Choral, the main notable difference with Listing 3 is that by removing assignments, we rely on method arguments for scoped variable declarations instead. The `updateEmails` method invokes the `getEmails` method, which fetches the emails from the `Server` by sending to it the `id` of the user and the timestamp (`ts`) of the last checkout and transmitting back the result of the extraction on the `serverDB`. Similar to Mini Choral, Mini ScalaLoci also lacks generics, an orthogonal language feature. The lack of generics, however, does not limit the expressiveness of the language to capture the correct topology of the system and communication between the `Server` and the `Client`. After obtaining the emails, we apply method `contextAwareUpdate`, which updates the email database of the client and proceeds to conditionally retrieve the attachments of the fetched emails.

5 Choreographies to Multitier, Multitier to Choreographies

We now define algorithms that translate programs in a Mini language to the other and vice versa. The reason for defining the following algorithms is to present evidence of the existence of a common root at the foundation of the two approaches. We show that the mechanised procedures for their reciprocal translation are relatively simple. In the remainder of this section, for brevity, we use the names Choral and ScalaLoci to indicate their Mini counterparts. We first present a translation algorithm from a Choral choreography to a ScalaLoci multitier application (Section 5.2). Afterwards, we show a translation algorithm from a ScalaLoci multitier application to a Choral choreography (Section 5.2).

Perspective translation Multitier and choreographic programming take different perspectives on what parts of the language are annotated with locations. In Choral, all literals are annotated by the role on which they operate, and the location of operators can be inferred by the location of their argument. ScalaLoci assigns peers to expressions, which are then written from the specified peer's perspective.

While in simple cases there is a direct correspondence between a value on the role `A` in Choral (`1@A`) and on a peer `A` in ScalaLoci (`on[A] { 1 }`), the difference is more obvious in compound expressions (`on[A] { 1 + 2 + 3 }` vs. `1@A + 2@A + 3@A`), where in ScalaLoci, only the whole expression is annotated but the literals are not, whereas in Choral, only the literals are annotated while the expression is not.

The translation algorithms perform such perspective change by grouping composed literals on the same Choral role into a ScalaLoci placed expression and, in the opposite direction, assigning the same Choral role to all literals in a ScalaLoci placed expression.

Further, we translate between ScalaLoci's way of defining peers and their topology as type members and Choral's way of defining roles as class parameters and their communication channels as class members.

Communication translation In ScalaLoci two peers communicate using `asLocal`. Given an expression `e` on peer `A`, the expression `on[B] { e.asLocal }` describes how peer `B` can access the value of `e`, implemented as a message with the value of `e` sent from `A` to `B`. In Choral, such communication is represented by invoking the `com` method of a directional communication channel, which takes a value on role `A` and returns it on role `B`.

The translation algorithms transform `asLocal` in ScalaLoci to an invocation of method `com` of the appropriate channel in Choral and vice versa.

5.1 From Choreographic Programming to Multitier Programming

Choral choreography classes to ScalaLoci multitier objects Algorithm 1 describes the translation of Choral choreography classes to ScalaLoci multitier objects. We decompose the class definition to be transformed into its identifier id, the roles $\overline{Role}$, the channel declarations $\overline{Channel}$, the field declarations $\overline{Field}$ and the method definitions $\overline{Method}$.

Each Choral role definition is translated to a ScalaLoci peer definition. Each channel `DiChannel@(A,B) ch_A_B` between two roles is translated to a single tie, e.g., a directed one-to-one tie, between two peers `@peer type A <: { type Tie <: Single[B] }`.

The translation of field definition from Choral to ScalaLoci is straightforward. In Choral, fields are introduced with a base type and the residing role, followed by the name of the field "id_{name}`@`(id_{role}) id_{type}". In ScalaLoci, fields are introduced as "`val` id_{name}: id_{type} `on` id_{role}". Similarly, method definitions are translated.

Finally, the algorithm returns a multitier object with the same name and the translated definitions as a body.

Choral choreography expressions to ScalaLoci multitier expressions Algorithm 2 describes the translation of Choral expressions to ScalaLoci: the algorithm matches on the different cases of Choral *Exp* terms and transforms each into the corresponding ScalaLoci code.

For sequencing $e_0; e_1$, both e_0 and e_1 are recursively transformed. If both subexpressions agree on their placement, e.g., $A = B$, the complete sequence is placed on the same peer. More interestingly, if the subexpressions are placed on different peers, we introduce a nested

Algorithm 1 Translation algorithm from Choral classes to ScalaLoci objects.

function *choral2loci(class)*
  "**class** id@($\overline{Role}$) { $\overline{Channel}$ $\overline{Field}$ $\overline{Method}$ }" $\leftarrow$ class
  $decls \leftarrow \{\ \}$
  for T $\leftarrow$ $\overline{Role}$ **do**
    $ties \leftarrow \{$ "**Single**[B]" | "DiChannel@(A, B) ch_A_B" $\in Channel \land T = A\ \}$
    $decls \leftarrow decls \cup \{$ "**@peer type** T <: { **type Tie** <: Any **with** $\overline{ties}$ }" $\}$
  end
  for "id_t@(A) id_n" $\leftarrow \overline{Field}$ **do**
    $decls \leftarrow decls \cup \{$ "**val** id_1 : id_0 **on** A" $\}$
  end
  for "id_t@(A) id $(\overline{id_{tn}@(A)\ id_{en}})$ { e }" $\leftarrow \overline{Method}$ **do**
    $e' \leftarrow choral2loci(e)$
    $decls \leftarrow decls \cup \{$ "**def** $id(\overline{id_{en} : id_{tn}})$: id_t **on** $A = \{e'\}$" $\}$
  end
  return "**@multitier object** id { $\overline{decls}$ }"
end

remote block for e'_0, which executes e'_0 on A and places the overall result of e'_1 on B. For the remote block we generate a capture clause for all method-local variables that are free in e_0.

The translations for identifiers, literals and instantiation is straightforward, placing the ScalaLoci expression on the peer according to the role specified in the Choral code. Further, the case for method invocation is similar since we assume that the receiver of a method invocation and its arguments are on the same role. This assumption is expressed by the *assert* statement in the algorithm and holds for every well-typed Mini Choral program. Selection does not exist in ScalaLoci. Hence, it is removed.

The case for branching makes a distinction similar to sequencing of whether the condition agrees to the branches regarding their placement, e.g., $A = B$. If they agree, the complete branching is placed on the same peer. Otherwise, we introduce a nested remote block for e'_0, which executes e'_0 on A and returns the result to B where the branches are placed. B then acts as a coordinator to decide which of the branches to execute.

Finally, we translate Choral's channel communication. For a channel from role B to A, we generate a ScalaLoci expression, which runs a nested remote block for e', which executes e' on B and returns the result to A.

5.2 From Multitier Programming to Choreographic Programming

ScalaLoci multitier objects to Choral choreography classes Algorithm 3 describes the translation of ScalaLoci multitier objects to Choral choreography classes. We decompose the multitier object to be transformed into its identifier id, the peer and tie declarations $\overline{Peer}$, the field declarations $\overline{Field}$ and the method definitions $\overline{Method}$.

Each ScalaLoci peer definition is translated to Choral role argument and each single tie between two peers is translated to a `DiChannel` between two peers `@(A,B)`.

The translation of field and method definitions from ScalaLoci to Choral is straightforward.

Finally, the algorithm returns a Choral class with the same name and the translated definitions as a body.

Algorithm 2 Translation algorithm from Choral expressions to ScalaLoci expressions.

function *choral2loci(expr)*
return match *expr* **with**
case "$e_0; e_1$" **with**
"on$[A]\{\ e'_0\ \}$" $\leftarrow$ $choral2loci(e_0)$
"on$[B]\{\ e'_1\ \}$" $\leftarrow$ $choral2loci(e_1)$
$captures \leftarrow freeVars(e_0) \cap currentMethodArguments$
if $A \neq B$ **then**
"on$[B]\{$ on$[A]$.run.capture$(\overline{captures})\ \{\ e'_0\ \}$.asLocal; $e'_1\ \}$"
else
"on$[B]\{\ e'_0; e'_1\ \}$"
end
case "id" **with**
$A \leftarrow roleOf(id)$
"on$[A]\{\ id\ \}$"
case "lit@A" **with**
"on$[A]\{\ lit\ \}$"
case "new id@$A\ (\overline{e})$" **with**
"on$[A]\{\ \overline{e'}\ \}$" $\leftarrow$ $\overline{choral2loci(e)}$
"on$[A]\{$ new $id(\overline{e'})\ \}$"
case "$e_0.id(\overline{e})$" **with**
"on$[A]\{\ e'_0\ \}$" $\leftarrow$ $choral2loci(e_0)$
"on$[B]\{\ \overline{e'}\ \}$" $\leftarrow$ $\overline{choral2loci(e)}$
assert $A = B$ *// receiver and arguments have the same role*
"on$[A]\{\ e'_0.id(\overline{e'})\ \}$"
case "ch.select(e)" **with**
"Unit"
case "if $(e_0)\ \{\ e_1\ \}$ else $\{\ e_2\ \}$" **with**
"on$[A]\{\ e'_0\ \}$" $\leftarrow$ $choral2loci(e_0)$
"on$[B]\{\ e'_1\ \}$" $\leftarrow$ $choral2loci(e_1)$
"on$[C]\{\ e'_2\ \}$" $\leftarrow$ $choral2loci(e_2)$
$captures \leftarrow freeVars(e_0) \cap currentMethodArguments$
assert $B = C$ *// branches have the same role*
if $A \neq B$ **then**
"on$[B]\{$ if (on$[A]$.run.capture$(\overline{captures})\ \{\ e'_0\ \}$.asLocal) $\{\ e'_1\ \}$ else $\{\ e'_2\ \}\ \}$"
else
"on$[B]\{$ if $(e'_0)\ \{\ e'_1\ \}$ else $\{\ e'_2\ \}\ \}$"
end
case "ch_B_A.com(e)" **with**
"on$[B]\{\ e'\ \}$" $\leftarrow$ $choral2loci(e)$
$captures \leftarrow freeVars(e) \cap currentMethodArguments$
"on$[A]\{$ on$[B]$.run.capture$(\overline{captures})\ \{\ e'\ \}$.asLocal $\}$"
end
end

ScalaLoci multitier expressions to Choral choreography expressions Algorithm 4 describes the translation of ScalaLoci expressions to Choral expressions. The algorithm matches on the different cases of ScalaLoci *Expr* terms and transforms each of them into the corresponding ScalaLoci code.

The translations for sequencing, identifiers, literals, instantiation and method invocation is straightforward, recursively transforming each subexpression.

In the case for branching, the translation needs to synthesise select expressions to implement knowledge of choice (recall Section 3.2). Hence, we collect all peers used in the

Algorithm 3 Translation algorithm from ScalaLoci objects to Choral classes.

function *loci2choral(module)*
  "`@multitier object` id { $\overline{Peer}$ $\overline{Field}$ $\overline{Method}$ }" ← *module*
  $decls \leftarrow \{\ \}$
  $roles \leftarrow \{\ \}$
  for "`@peer type` A `<:` { `type Tie <: Any with` $\overline{ties}$ }" ← *Peer* **do**
    $roles \leftarrow roles \cup \{ A \}$
    for "`Single`[B]" ← *ties* **do**
      $decls \leftarrow decls \cup \{$ "DiChannel@(A,B) ch_A_B" $\}$
    end
  end
  for "`val` id_1: id_0 `on` A" ← *Field* **do**
    $decls \leftarrow decls \cup \{$ "id_0@(A) id_1" $\}$
  end
  for "`def` $id(\overline{id_{en} : id_{tn}})$: id_t `on` $A = \{ e \}$" ← *Method* **do**
    $e' \leftarrow loci2choral(e)$
    $decls \leftarrow decls \cup \{$ "id_t@(A) $id(\overline{id_{tn}@(A)\ id_{en}})$ { e' }" $\}$
  end
  return "`class` id@($\overline{roles}$) { $\overline{decls}$ }"
end

branches and create select statements for all channels between those peers for both branches.

Finally, we translate ScalaLoci's nested remote blocks. For a remote expression placed on A that executes e on B, we generate a Choral channel communication that transfers the value of e from B to A.

6 A Unified Perspective

Although choreographic and multitier programming evolved in dissimilar ways, their cores—represented by our two Mini languages—are close enough to let us define in Section 5 straightforward translation algorithms in both directions and show the core features of both approaches isomorphic.

Besides the more abstract purpose to present evidence of the closeness of the two approaches, our translation algorithms are also directly useful in practice. Translating Choral to ScalaLoci code enables the reuse of ScalaLoci's middleware for Choral. In general, translating to multitier programs is interesting because we can leverage the possibility of compiling to different technologies.

Translating ScalaLoci to Choral code enables synthesising the choreography of the multitier program. Making the protocol manifest supports both manually checking what communications take place as well as automatic analyses (e.g., security).

We believe that both the multitier and choreographic research areas can greatly benefit from cross-fertilisation and transfer of concepts already developed in one but lacking in the other. As a glimpse of this fact, we dedicate Section 6.1 to describe some advanced features present in only one of the two languages (Choral, ScalaLoci) and outline how they could be integrated into the other in the future. We conclude this section by widening our scope on the category of multiparty language in Section 6.2. We give an (incomplete) overview on other languages that are neither multitier nor choreographic but share common traits that can classify them as multiparty ones. We consider those languages valuable additions to the

Algorithm 4 Translation algorithm from ScalaLoci expressions to Choral expressions.

function $loci2choral(expr)$
  return match $expr$ **with**
    case "on$[A]\{\ e_0;\ e_1\ \}$" **with**
      $e'_0 \leftarrow loci2choral($"on$[A]\{\ e_0\ \}$"$)$
      $e'_1 \leftarrow loci2choral($"on$[A]\{\ e_1\ \}$"$)$
      "$e'_0;\ e'_1$"
    case "on$[A]\{\ id\ \}$" **with** "id"
    case "on$[A]\{\ lit\ \}$" **with**
      "lit@A"
    case "on$[A]\{$ new $id(\overline{e})\ \}$" **with**
      $\overline{e' \leftarrow loci2choral(\text{"on}[A]\{\ e\ \}\text{"})}$
      "new id@$A(\overline{e'})$"
    case "on$[A]\{\ e_0.id(\overline{e})\ \}$" **with**
      $e'_0 \leftarrow loci2choral($"on$[A]\{\ e_0\ \}$"$)$
      $\overline{e' \leftarrow loci2choral(\text{"on}[A]\{\ e\ \}\text{"})}$
      "$e'_0.id(\overline{e'})$"
    case "on$[A]$ { if (e_0) { e_1 } else { e_2 } }" **with**
      $e'_0 \leftarrow loci2choral($"on$[A]\{\ e_0\ \}$"$)$
      $e'_1 \leftarrow loci2choral($"on$[A]\{\ e_1\ \}$"$)$
      $e'_2 \leftarrow loci2choral($"on$[A]\{\ e_2\ \}$"$)$
      $peers \leftarrow peersIn(e'_1) \cup peersIn(e'_2)$
      $channels \leftarrow \{$ "ch_A_B" $\mid B \in peers \wedge A$ has tie to $B\ \}$
      $thenSelects \leftarrow \{$ "$c.select($Choice@A.THEN$)$" $\mid c \in channels\ \}$
      $elseSelects \leftarrow \{$ "$c.select($Choice@A.ELSE$)$" $\mid c \in channels\ \}$
      "if (e'_0) { $\overline{thenSelects}$; e'_1 } else { $\overline{elseSelects}$; e'_2 }"
    case "on$[A]\{$ on$[B]$.run.capture$(\overline{captures})$ { e }.asLocal }" **with**
      $e' \leftarrow loci2choral($"on$[B]\{e\}$"$)$
      "ch_B_A.com(e')"
  end
end

Feature	Choral	ScalaLoci
Distributed Data Structures	✓	×
Dynamic Topologies	×	✓
Higher-Order Composition	✓	×
Races	–	✓
Fault tolerance	✓	✓
Asynchrony	✓	✓

Table 1 Overview of the feature comparison of choreographic and multitier programming.

multiparty category and subject of future research akin to this work.

6.1 Feature Comparison

We now discuss a few features that are important for concurrent and distributed programming. Our discussion is summarised in Table 1, which shows which features are present in Choral and ScalaLoci, respectively (the – means partial presence, explained in the relevant paragraph where we discuss the feature). The first four features have evolved separately and give potential for cross-fertilisation, whereas the last two are important features that have been dealt in both worlds (yet separately).

Distributed Data Structures

The `@(R1, ..., Rn)` type notation supported in Choral specifies the distribution of classes and objects over roles. This is true also without taking into account communication. As an example, let us consider the `BiPair` class below, which implements an incarnation of a `Pair` class where the two values (referred as `left` and `right`) of the pair belong to different roles.

```
class BiPair@(A,B)<L@X, R@Y> {
  private L@A left;
  private R@B right;
  public BiPair(L@A left, R@B right) { this.left = left; this.right = right; }
  public L@A left() { return this.left; }
  public R@A right() { return this.right; }
}
```

As its Java counterpart, also `BiPair` is parametric with respect to its contents: we use parameters `L` and `R` to capture the type of the left and right components of the pair. Then, by specifying that `L` is owned by one role `X` and `R` is owned by another role `Y`, we indicate that the two values in the pair must be at different roles (and they can capture different data types, e.g., `String` and `Integer`). Indeed, adopting the same interpretation of Java generics, Choral interprets role parameter binders so that the first appearance of a parameter is a binder, while subsequent appearances of the same parameter are bound—hence, given that the declaration of type parameters `<...>` limits the scope of the of role parameters `X` and `Y`, we are indicating that they cannot coincide. For completeness, we include in the definition of the `BiPair` class its fields (`left` and `right`, respectively located at `A` and `B`), a constructor, and the traditional accessors.

Besides showing the basic feature of inherent distribution supported by the Choral type system, the example of `BiPair` is useful to illustrate that, also without considering communications, Choral offers support in defining programs where the data at some role needs to correlate with data at another, e.g., as in the case of distributed authentication tokens.

Similar to Choral, in ScalaLoci, we use parameters `L` and `R` to capture the type of the left and right components of the pair. Corresponding to Choral's roles definition, we define an `A` and a `B` peer type. We then specify that `L` is placed on a peer `A` and `R` is placed on a peer `B`:

```
@multitier trait BiPair[L, R] {
  @peer type A
  @peer type B

  val left: L on A
  val right: R on B
}
```

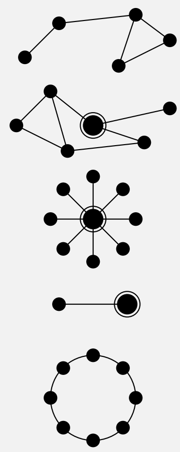
While we can define distributed data structures similar to Choral, they have to be composed at compile-time limiting their usability because of ScalaLoci's lack of higher-order composition.

Dynamic Topologies and Homogenous Behaviours

A feature of ScalaLoci that is not covered in its Mini variant is the possibility for peer types to abstract over multiple peer instances of the same type, e.g., a master-worker architecture where a single master can connect to an arbitrary number of homogeneous (i.e., with the

Listing 8 Distributed Architectures.

```
@multitier object P2P {
  @peer type Peer <: { type Tie <: Multiple[Peer] }
}
@multitier object P2PRegistry {
  @peer type Registry <: { type Tie <: Multiple[Peer] }
  @peer type Peer <: { type Tie <: Optional[Registry] with Multiple[Peer] }
}
@multitier object MultiClientServer {
  @peer type Server <: { type Tie <: Multiple[Client] }
  @peer type Client <: { type Tie <: Single[Server] with Single[Node] }
}
@multitier object ClientServer {
  @peer type Server <: { type Tie <: Single[Client] }
  @peer type Client <: { type Tie <: Single[Server] with Single[Node] }
}
@multitier object Ring {
  @peer type Node <: { type Tie <: Single[Prev] with Single[Next] }
  @peer type Prev <: Node
  @peer type Next <: Node
  @peer type RingNode <: Prev with Next
}
```



same behaviour) worker nodes. Such a feature also enables dynamic topologies where peers can join and leave the system at run time. A variable number of peer instances is expressed in ScalaLoci's peer specification by not using a `Single` tie but an `Multiple` or an `Optional` tie, i.e., an arbitrary number or at most one remote peer of a given type can connect, respectively.

Listing 8 shows the definitions for different topologies with their iconification on the right. The `P2P` module defines a `Peer` that can connect to arbitrary many other peers. The `P2PRegistry` module adds a central registry, to which peers can connect. The `MultiClient-Server` module defines a client that is always connected to a single server, while the server can handle multiple clients simultaneously. The `ClientServer` module specifies a server that always handles a single client instance. For the `Ring` module, we define a `Prev` and a `Next` peer. A `RingNode` itself is both a predecessor and a successor. All `Node` peers have a single tie to their predecessor and a single tie to their successor.

ScalaLoci allows to abstract over different peer instances of the same type and uniformly receive values from multiple connected remote peers, `asLocalFromAll` returns a sequence that contains the remote values from the different peers. Yet, a specific peer instance `client` can be selected via `on(client).run { ... }.asLocal` (using the `client` value referencing a peer instance) instead of `on[Client].run { ... }.asLocal` (using the `Client` peer type). The handlers `remote[Client].join foreach { ... }` and `remote[Client].leave foreach { ... }` can be used to react to dynamic changes in the topology of the running multitier system.

Denièlou and Yoshida [13] developed a theory for choreographies with homogeneous roles and dynamic topologies by allowing choreographies to be parametrised (also) in collections of roles. Plans for supporting for this feature in Choral are discussed in [16, §7]. In this extension, prefixing a role parameter declaration with $*$, as in `*Clients`, specifies that this is a collection of roles. Types are extended with products indexed over collections of role using a syntax similar to Java `for-each` blocks. For instance, the type `forall(Client: Clients) String@Client` represents a "tuple" with a `String` for each role in the collection `Clients`. We can write a scatter-gather channel over a star topology (cf. `MultiClientServer`) as follows:

```
abstract class StarChannel@(Server,*Clients) {
  forall(Client : Clients) {SymChannel@(Server,Client)} star;
  forall(Client : Clients) {String@Client} scatter(String@Server m);
  String@Server gather( forall(Client : Clients) {String@Client} ms);
}
```

Method gather of StarChannel is then translated to ScalaLoci's primitive asLocalFromAll and vice versa. A further extension discussed in [16, §7] is the introduction of existential quantification over roles in role collections. For instance, with(Client: Clients) String@(Client) represents a string at some role in the collection Clients. We can extend the example above to support any-cast communication as follows:

```
abstract class StarChannel@(Server,*Clients) {
  /* ... */
  with(Client : Clients) {String@(Client)} any( String@Server m );
  String@Server any( with(Client : Clients) {String@(Client)} m );
}
```

Method any of StarChannel is then translated to ScalaLoci's on(c).run { ... } and vice versa.

Higher-Order and First-Class Multiparty Programs

We classify "higher-order" a multiparty language where multiparty components (objects, functions) are values that can be passed as arguments.

Choral is higher-order because methods can accept choreographic objects with multiple roles as parameters. In Choral, Channels are one of the most basic examples of the usage of the higher-order feature. For example, we can pass a DiChannel as an argument.

```
class MyClass@(A,B){
  void passValue( DiChannel@(A,B) ch ){
    ch.com< Integer >( 5@B );
  }
}
```

In the example, the method passValue takes as input the choreographic object DiChannel and, by invoking its com method, we execute the protocol needed to send the data (5@B) between the two roles.

ScalaLoci does not support higher-order composition (no multitier objects as values or dynamic multitier object storage) but at least supports statically-composed modules [39]. The following snippet shows the declaration of a ClientServer multitier module that is parameterised over a Client and a Server peer. The module uses the monitoring functionality provided by the Monitoring multitier module, which is parameterised over a Monitor and a Monitored peer. The Monitoring module is instantiated by mon inside ClientServer. The ClientServer module identifies the Client peer with the Monitored peer and the Server peer with the Monitor peer and defines their ties accordingly:

```
@multitier trait Monitoring {
  @peer type Monitor { type Tie <: Single[Monitored] }
  @peer type Monitored { type Tie <: Single[Monitor] }
}

@multitier object ClientServer {
  @multitier object mon extends Monitoring

  @peer type Client <: mon.Monitored { type Tie <: Single[mon.Monitor] with Single[Server] }
  @peer type Server <: mon.Monitor { type Tie <: Single[mon.Monitored] with Single[Client] }
}
```

Races

Both Choral and ScalaLoci support programs with races to some degree. We distinguish two prototypical scenarios: races among producers and races among consumers.

To program a race among multiple producers in ScalaLoci, we can simply retrieve the values from all remote producers via `asLocalFromAll` and pick the first one that becomes available via `Future.firstCompletedOf` as shown in the example below:

```
Future.firstCompletedOf(
  on[Producer].run { generateValue() }.asLocalFromAll map {
    case (producerPeerInstance, value) => value map { (producerPeerInstance, _) }
  })
```

It is not possible to program a race among multiple consumers in ScalaLoci.

In Choral, it is possible to implement protocols with races among producers and among consumers provided their number is statically fixed. For instance, below is the type for a choreography where two producers rage to send a message to a consumer.

```
interface ProducerRace(Producer1,Producer2,Consumer) {
 Message@Consumer run(Message@Producer1 m1, Message@Producer2 m2);
}
```

The constraint that the number of roles must be statically fixed is related to the inability of Choral to capture dynamic topologies and, as discussed above, is solved by adding collections of roles to the language. In the case of consumer races, another limitation is that the Choral type system is not powerful enough express (and enforce) their presence. Consider a situation where two consumers race to receive a message from a single producer. In Choral, this protocol can implement the following interface:

```
interface ConsumerRace(Producer,Consumer1,Consumer2) {
 BiPair@(Consumer1,Consumer2)<Optional<Message>,Optional<Message>> run(Message@Producer m);
}
```

However, the return type of `run` does not guarantee that exactly one consumer receives the message: implementations that deliver the message to both or neither respect the type. As discussed in [16, §7], we can write a precise type if we extend Choral with existential quantification over roles as shown in the example below.

```
interface ConsumerRace@(Producer, Consumer1, Consumer2) {
  with (C : [Consumer1, Consumer2]){ Message@C } run(Message@Producer m);
}
```

Fault Tolerance

In ScalaLoci, remote values whose computation or transmission to the local peer instance fail result in a future that is completed with a failure value. Thus, user code can detect a failed remote access and decide how to react appropriately. Failed futures can be handled using the usual operators on futures, e.g., `recover`:

```
on[Client].run { generateValue() }.asLocal recover { case _ => generateOtherValue() }
```

Like other aspects of communication in Choral, the language does not commit to specific interaction patterns: programmers can implement their own strategies e.g. using exceptions or returning errors.

Asynchrony

For the sake of exposition, we presented multiparty programs using communication APIs as if they were blocking and designed the Mini variants of both Choral and ScalaLoci as synchronous. ScalaLoci promotes an asynchronous approach: the preferred variant of accessing remote values via `asLocal` in ScalaLoci creates a future to account for network delay and potential communication failure. On the other hand, Choral is agnostic with regards to communication models: programmers can import libraries of channels or implement their own. For instance, a communication model similar to ScalaLoci's `asLocal` is offered by the following interface:

```
interface AsyncDiChannel@(Sender, Receiver)<T@X> {
  <S@Y extends T@Y> Future@Receiver<S> com(Promise@Sender<S>);
}
```

6.2 Other Multiparty Languages

For the future we envision further cross-fertilisation between multiparty languages, and that the class of multiparty languages might get larger. We mention a few approaches that might contribute to this.

Software architectures [14, 33] are about organising software systems into well-studied patters that comprise components and their connections organised in a certain configuration. Architecture description languages (ADL) [26] specify software architectures and the constraints among the architecture components. Different from choreographic and multitier programming, ADLs usually specification languages separate from the implementation. An exception is ArchJava [1] which support specifying a software architecture and enforcing its constraints together with the implementation. Regarding cross-fertilisation, ADLs come equipped with powerful analysis, code synthesis, and runtime-support tools as well as model checkers, which can be also used in multitier and choreographic scenarios to enforce different aspects of correctness.

Partitioned global address space languages (PGAS) [12] are often used in the domain of high-performance computing. The main abstraction is a global memory address space where logical partitions are assigned to processes to maximize data locality. X10 [7] features explicit fork/join operations and provides a sophisticated dependent type system [6] to model the *place* (the heap partition) a reference points to. PGAS languages, similar to multitier and choreographic languages reduce the boundaries between hosts in a distributed system.

7 Conclusion

Choreographic and multitier languages have developed independently, leading to a number of research achievement carried out within two vibrant but separate research communities [2, 28, 40]. In this paper, we discussed the fundamental nature of the programming paradigms based on these languages, isolating the core difference between them. We then showed that, under the cover of syntactic variance, the two approaches are similar enough to be related and to reason about potential cross-fertilisation. Our observations offer a platform for future joint work between the respective communities.

References

1 Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on*

Software Engineering, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM. doi:10.1145/581339.581365.

2 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. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

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

4 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

5 Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998. doi:10.1007/BFb0053547.

6 Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 11–22, New York, NY, USA, 2008. ACM. doi:10.1145/1345206.1345211.

7 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094852.

8 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006. doi:10.1007/978-3-540-74792-5_12.

9 Ezra E. K. Cooper and Philip Wadler. The RPC calculus. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 231–242, New York, NY, USA, 2009. ACM. doi:10.1145/1599410.1599439.

10 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2016. doi:10.1007/978-3-319-39570-8_8.

11 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

12 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4), May 2015. doi:10.1145/2716320.

13 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 435–446. ACM, 2011. doi:10.1145/1926385.1926435.

14 David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994. Accessed 2020-05-05. URL: http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf.

15 Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, pages 22–40. Springer, 2018. doi:10.1007/978-3-030-02671-4_2.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. URL: https://arxiv.org/abs/2005.09520, arXiv:2005.09520.

17 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI '73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. Accessed 2020-05-05. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.

18 Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, volume 6536 of *Lecture Notes in Computer Science*, pages 55–75. Springer, 2011. doi:10.1007/978-3-642-19056-8_4.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008. URL: http://doi.acm.org/10.1145/2827695, doi:10.1145/2827695.

20 Intl. Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

21 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proc. of ASPLOS*, pages 517–530, 2016.

22 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015.

23 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In *SAC*, pages 437–443. ACM, 2017.

24 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In *FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016.

25 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. of ASPLOS*, pages 329–339, 2008.

26 Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. doi:10.1109/32.825767.

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

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

29 Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 286–295. IEEE Computer Society, 2004. doi:10.1109/LICS.2004.1319623.

30 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

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

32 Peter W. O'Hearn. Experience developing and deploying concurrency analysis at facebook. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2018. doi:10.1007/978-3-319-99725-4_5.

33 Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. doi:10.1145/141874.141884.

34 Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In Atsushi Igarashi, editor, *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS '16, pages 377–397, Berlin, Heidelberg, November 2016. Springer-Verlag. doi:10.1007/978-3-319-47958-3_20.

35 Bob Reynders, Frank Piessens, and Dominique Devriese. Gavial: Programming the web with multi-tier FRP. *The Art, Science, and Engineering of Programming*, 4(3):6:1–6:32, February 2020. doi:10.22152/programming-journal.org/2020/4/6.

36 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 975–985. ACM, 2006. doi:10.1145/1176617.1176756.

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

38 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):129:1–129:30, October 2018. doi:10.1145/3276499.

39 Pascal Weisenburger and Guido Salvaneschi. Multitier modules. In Alastair F. Donaldson, editor, *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:29, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10795, doi:10.4230/LIPIcs.ECOOP.2019.3.

40 Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Computing Surveys*, 53(4), September 2020. doi:10.1145/3397495.