Contents

Listings

List of Figures

CHAPTER 1

Chor: a Framework for Choreographic Programming

Fabrizio Montesi ⟨`fmontesi@itu.dk`⟩ **– IT University of Copenhagen**

1.1 Introduction

In Chapter **??** we have presented the Choreography Calculus, a calculus in which systems can be developed by writing a choreography, verifying it against protocol specifications given as global types [7], and then projecting a correct endpoint implementation. This methodology is depicted below:

Building on the Choreography Calculus, in this Chapter we present Chor, a prototype framework for Choreographic Programming. Chor offers a programming language, based on choreographies, and an Integrated Development Environment (IDE) developed as an Eclipse plugin [5] for the writing of programs. All code is open source and can be consulted at the Chor website [4], along with an introductory video tutorial. Chor is available also on the Eclipse Marketplace, the official distribution channel for Eclipse-based software.

In the development methodology suggested with Chor, depicted in Figure 1.1, developers can first use our IDE to write protocol specifications and choreographies. The programmer is supported by on-the-fly verification which takes care of checking (i) the syntactic correctness of program terms and (ii) the type compliance of the choreography wrt the protocol specifications, using our typing discipline from § **??**. Program errors are reported using syntax highlighting, allowing for an interactive programming experience.

Once the global program is completed, developers can automatically project it to an endpoint implementation. Endpoint implementations are given in the Jolie programming

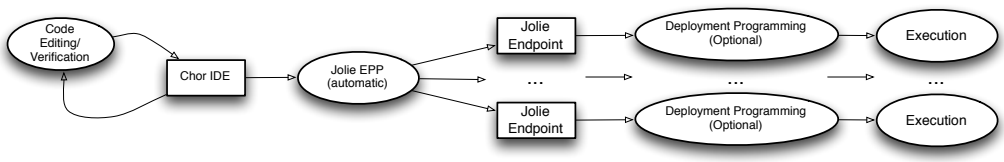

Figure 1.1: Chor, development methodology.

language [9]. Nevertheless, Chor is designed to be extended to multiple endpoint languages: potentially, each process in a choreography could be implemented with a different endpoint technology. We plan future extensions to support projecting endpoints to, e.g., Java, C#, or WS-BPEL [10].

Each Jolie endpoint program comes with its own deployment information, given as a term separated from the code implementing the behaviour of the projected process. This part can be optionally customised by the programmer, which can be useful for adapting the endpoint programs to some specific network or communication technology. For example, programs generated by Chor can be integrated with web applications by using the `http` extension presented in Chapter **??**.

Finally, the Jolie endpoint programs can be executed with our extension of the Jolie interpreter presented in Chapter **??**. As expected, they will implement the originating choreography.

1.1.1 Contributions

This Chapter provides the following contributions.

A programming language for choreographies. We present a language implementation for programming concurrent systems with choreographies (§ 1.2). Our language natively supports the development of multiparty sessions and their type checking against protocol specifications, following the model in Chapter **??**. Programs in Chor are written in an IDE that supports on-the-fly type checking.

Endpoint Projection. A choreography program written in Chor can be automatically compiled to a set of Jolie programs that will implement the processes in the originating choreography (§ 1.3). We refer to this compilation procedure, as in the other Chapters, as Endpoint Projection (EPP). The key to our implementation of projection is using correlation sets (from Chapter **??**) for supporting the execution of multiparty sessions.

Evaluation. We evaluate Chor with a series of real-world examples, e.g., authentication protocols and a use case from the Ocean Observatories Initiative (OOI) [11] (§ 1.4).

1.2 Language

In this section, we discuss the syntax of the Chor language.

1.2.1 Program structure

The syntax of programs is reported in Figure 1.2. A Chor program starts with the statement **`program`** `id`, which declares a name for the program with the identifier `id` (giving names to programs is reserved for future use in project management features). A program is then composed by two parts: a $Preamble$ and a $Body$.

The program preamble defines the protocols (* indicates that a nonterminal may be repeated) and public names that will be used in the choreography of the program. Each protocol definition is identified in the program by a name `g`, which is associated to a global type G. Public names, which are the shared names from our Choreography Calculus (see

Program	::=	**program** id; *Preamble* *Body*	*(program)*
Preamble	::=	*Protocol** *Public**	*(preamble)*
Protocol	::=	**protocol** g { *G* }	*(protocol)*
Public	::=	**public** a : g	*(public channel)*
Body	::=	*Def** **main** { *C* }	*(body)*

Figure 1.2: Chor, syntax of programs.

G	::=	A -> B : op(*U*) ; *G*	*(com)*
	|	A -> B : {op$_i$ (U_i) : G_i}$_{i \in I}$	*(branch)*
	|	g	*(call)*
	|	**end**	*(end)*
U	::=	**void** | **bool** | **string** | **int**	*(data type)*
	|	g@C	*(delegation type)*

Figure 1.3: Chor, syntax of protocols.

§ **??**), are used to start sessions in a choreography. Each public name, identified by a name a, is associated to a protocol g that the session started through a will be expected to follow.

The program body gives a list of procedure definitions Def^* and a main procedure **main** containing a choreography C, which is the program entry-point for execution.

1.2.2 Protocols

Protocols are expressed in terms of global types, whose syntax is reported in Figure 1.3. A *(com)* global type A -> B : op(*U*) ; *G* specifies a communication from role A to role B through operation op, where the content of the message is required to have type U. Then, the global type proceeds as G. A *(branch)* global type is similar, but A can now choose among a set of different operations, each one with a corresponding carried type U_i and continuation G_i. In term *(call)*, we indicate that the global type proceeds as the global type associated to protocol g. Type **end** indicates termination of a protocol; it is automatically inserted by our implementation and it thus omitted in programs.

Remark 1.2.1 (Global types in Chor). Global types in Chor are a variant of those presented for the Choreography Calculus in § **??**. Operations are simply the labels l used in § **??** for handling branchings; here, we chose to call labels operations to be nearer to the terminology of implemented languages for service-oriented computing. The only major difference wrt § **??** is that here we require the specification of an operation also for normal communications with no branching. This does not change the expressivity of global types in any way since we could, e.g., retain typical global types by adopting a standard operation name for non-branching communications. Our syntax allows us to give global types that

Def	::=	**define** d($\widetilde{\mathtt{p}}$)($\tilde{D}$) { C }	*(def)*
D	::=	k:g[p[A]]	*(def param)*
C	::=	η ; C	*(interaction)*
	|	τ ; C	*(local action)*
	|	**if**(e)@p C_1 **else** C_2	*(cond)*
	|	d($\widetilde{\mathtt{p}}$)($\widetilde{\mathtt{k}}$)	*(call)*
	|	{ C }	*(block)*
	|	**end**	*(end)*
η	::=	p[A] **start** q[B] : a(k); C	*(start)*
	|	p.e -> q.x : op(k) ; C	*(com)*
	|	p -> q : op(k(k')) ; C	*(del)*
τ	::=	**local@**p(x = e)	*(assign)*
	|	**ask@**p(e, x)	*(ask)*
	|	**show@**p(e)	*(show)*

Figure 1.4: Chor, syntax of choreographies.

will relate directly to our generated Jolie code, in which communications always specify an operation. □

1.2.3 Choreographies

Procedure definitions and choreographies have similar syntax to that presented in the Choreography Calculus from Chapter **??**; it is reported in Figure 1.4.

Procedures. Term *(def)* defines a procedure d with a body C, declaring the processes $\widetilde{\mathtt{p}}$ and the sessions $\widetilde{\mathtt{k}}$ that are used inside C. Each session is typed with a protocol g and the processes that have to implement it playing their respective roles in C.

Interactions. A choreography C defines the behaviour of a system of processes. In term *(start)*, the active processes $\widetilde{\mathtt{p}}$ start the (fresh) service processes $\widetilde{\mathtt{q}}$ by synchronising on public channel a to start a new session k; each process is annotated with the role A it plays in the session. In term *(com)*, the sender process p sends the evaluation of its local expression e to the receiver process q, which stores the received value in its local variable x, through operation op on session k; e and x can be omitted when unnecessary. Term *(del)* implements a session delegation: process p delegates to process q, through session k, its role in session k'.

Remark 1.2.2 (Process role annotations). Differently than in the Choreography Calculus in **??**, Chor requires processes to be annotated with the role they play in a session only in session starts and the declaration of procedure parameters. Process roles are automatically inferred in all other terms. □

Local actions. We extend the Choreography Calculus from **??** with actions that can be performed locally by a process. Term *(assign)* implements a local variable assignment, where process `p` assigns the evaluation of its local expression `e` to its local variable `x`. Terms *(ask)* and *(show)* are used for interacting with a user. In term *(ask)*, process `p` asks the user to input a value for variable `x` by displaying message `e` with a message dialog. In term *(show)*, process `p` shows the user a message `e`.

Other terms. In term *(cond)*, process `p` evaluates an internal condition `e` and decides whether the system should proceed as choreography C_1 or C_2. Term *(call)* implements a procedure call, passing the processes $\widetilde{p}$ and sessions $\widetilde{k}$ as parameters. Term *(block)* is the standard block construct, used for explicitly grouping statements. Term *(end)* denotes termination; as in global types, it is automatically used by our implementation and is omitted in programs.

Example 1.2.3 (Chor program example)**.** We exemplify the syntax of Chor by giving a simple program:

```
1  program simple;
2
3  protocol SimpleProtocol { C -> S: hi( string ) }
4
5  public a: SimpleProtocol
6
7  main
8  {
9    client[C] start server[S] : a( k );
10   ask@client( "[client] Message?", msg );
11   client.msg -> server.x : hi( k );
12   show@server( "[server] " + x )
13 }
```

Program `simple` above starts by declaring a protocol `SimpleProtocol`, in which role `C` (for client) sends a string to a role `S` (for server) through operation `hi`. In the choreography of the program, process `code` and a fresh service process `server` start a session `k` by synchronising on the public channel `a`. Process `client` then asks the user for an input message and stores it in its local variable `msg`, which is then sent to process `server` through operation `hi` on session `k`, implementing protocol `SimpleProtocol`. Finally, process `server` displays the received message on screen. □

1.3 Implementation

In this section we comment the major aspects of the implementation of Chor.

1.3.1 Chor IDE

Chor comes with an IDE (Integrated Development Environment) developed as an Eclipse [5] plugin based on the Xtext framework [14]. The IDE offers three main components: a code

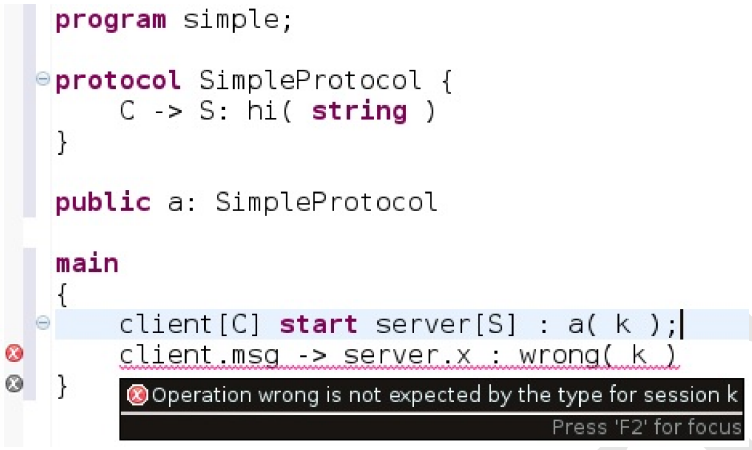

Figure 1.5: Chor, example of error reporting.

editor for Chor programs; an on-the-fly type checker; and an automated endpoint projection (EPP) implementation for obtaining executable code from a Chor program.

The code editor takes care of checking that a program follows our syntax and offers basic refactoring capabilities, e.g., the name of a protocol can be modified and then the change is automatically reflected in all references to the protocol in the rest of the program.

Our type checker is an implementation of the typing discipline in § **??**. Since type checking is efficient (it has polynomial computational complexity) we run it on the fly, i.e., whenever the program is modified. The programmer is then interactively notified of mistakes, such as not implementing a protocol correctly in a session. This is visually represented by the typical red underlining of errors in terms. We give an example of error reporting in Figure 1.5, in which session `k` uses operation `wrong` instead of the correct operation `hi` specified by the protocol. In the figure, we also show the entire error message that the user can access through the standard Eclipse user interface.

The Endpoint Projection (EPP) procedure in Chor targets our extension of the Jolie language from Chapter **??**. The choice of Jolie has several reasons: (i) Jolie offers constructs similar to those of our endpoint calculus, e.g., replicated services and input choices, making part of our EPP straightforward; (ii) the support for programming multiparty sessions based on correlation sets, which we used for implementing the sessions programmed in Chor; and (iii) Jolie supports a wide range of compatibility with other technologies.

1.3.2 Deployment

By default, our endpoint programs will operate on top of TCP/IP sockets. However, since Jolie also supports other communication technologies – e.g. local memory IPC and Bluetooth – and data formats – e.g. HTTP (cf. Chapter **??**) and SOAP [2] – programmers may customise deployment information of each endpoint. Hence, some endpoints may communicate over, e.g., HTTP, while others, e.g., using fast binary data formats. Additionally, different endpoints may be deployed in different machines and/or networks, giving developers freedom on choosing the degree of distribution of the system, from a single machine (e.g., multi-core systems) to a completely distributed setting (one endpoint program per machine).

Notably, customising the deployment of an endpoint program does not necessarily require updating the code of the others. Supporting this flexibility has required a careful implementation of session starts (rule $\lfloor^{\mathrm{P}}|_{\mathrm{START}}\rceil$ from our endpoint model in § **??**), which are coordinated by special "start services". The (endpoint projections of the) active processes willing to start a session contact the appropriate start service. Then, the start service spawns the (projections of the) service processes by calling the external services that implement them. In such message exchanges, which follow a variant of the standard Two-Phase Commit protocol [6], each endpoint informs the start service of the *binding information* (e.g., IP address and data format) on which the endpoint can be reached. Finally, the start service informs all participants about all necessary bindings, so that each party can dynamically update its references to the others (e.g., socket, bluetooth, or local inter-process connections).

Another key feature is that our implementation of message queues for sessions is based on the correlation mechanism developed in Chapter **??**. Specifically, for each session/-role pair that a process will play during execution we generate a corresponding correlation set. We have extended the implementation of Jolie to handle a separate message queue for each different correlation set, so to handle messages in different sessions concurrently as required by our endpoint model in § **??**. Afterwards, the programmer can customise correlation for each deployment artifact. For instance, some processes may identify sessions using HTTP cookies (using our configuration parameters for HTTP in Jolie from Chapter **??**), while others may use SOAP headers (as in the WS-Addressing specifications [3]).

Example 1.3.1 (Endpoint Projection in Chor)**.** We give an example of EPP by reporting a snippet of the code generated for process `server` from Example 1.2.3:

```
1  main
2  {
3    _start();
4    csets.tid = new;
5    _myRef.binding << global.inputPorts.MyInputPort;
6    _myRef.tid = csets.tid;
7    _start_S@a(_myRef)(_sessionDescriptor.k);
8    k_C << _sessionDescriptor.k.C.binding;
9    hi(x);
10   showMessageDialog@SwingUI("[server] " + x)()
11 }
```

The Jolie code above for process `server` waits to be started by receiving an input on operation `_start` (which acts as replicated). This starts the commit protocol for creating session `k`. In Lines 4–6, the process initialises its correlation value for the session (`csets.tid`), and then stores its binding information (e.g., its IP address) and correlation value for the session in variable `_myRef`. In Line 7, the process completes the session start protocol by sending its binding and correlation information to the service responsible for synchronising the processes involved in the creation of the session (service `a`). Still in Line 7, we receive in response a session descriptor for session `k`, which contains the binding information for reaching the other processes in the session. In Lines 9–10, we receive the message on operation `hi` from the client and display it on screen as indicated by the

choreography. □

1.3.3 Delegation

Session delegation is a nontrivial mechanism at the level of endpoint implementation. The main concern lies in updating channel references (bindings). For instance, assume that a session k has some process participants, say p and $\widetilde{q}$. Suppose now that p delegates its role in k to another process r through a different session. In such a situation, all the processes in $\widetilde{q}$ need their external references to be updated for reaching r instead of p when communicating with the session/role pair delegated by p. In our formal endpoint model (see § **??**) this necessity is completely abstracted away by the synchronisations on the centralised message queues (one per session). However, in our implementation of Chor, message queues are completely distributed: each process owns a message queue which must be reached explicitly by other process, following our model for correlation-based sessions from Chapter **??**. In [8] the authors present a survey of possible solutions to this problem in asynchronous scenarios[1]. The main challenge is that the processes in $\widetilde{q}$ may send messages to process p before getting notified of the delegation; in [8], this issue is solved by making process p resending these messages to process r, adding extra communications. In our EPP implementation, instead, each process in $\widetilde{q}$ knows when the delegation will happen since we have that information from the originating choreography. Hence, we program the processes in $\widetilde{q}$ to wait for receiving the updated binding information for reaching process r before proceeding in the session, guaranteeing that we are always in the optimal case where no messages are sent to the wrong recipient.

1.4 Examples

In this section, we discuss some examples that we have implemented using the Chor language. Specifically, our examples below show how to deal with five typical aspects of distributed systems: multiparty sessions, session interleaving, service selection, delegation, and recursive protocols. More examples, including the implementations of the examples in § **??**, are available at the Chor website [4].

1.4.1 Multiparty Sessions

We show how to address multiparty sessions by implementing a protocol inspired by the OpenID specifications for distributed authentication [12]:

```
1 program openid;
2
3 protocol OpenID {
4   U -> RP: username( string );
5   RP -> IP: username( string );
6   U -> IP: password( string );
7   IP -> RP: {
```

[1] The work in [8] actually deals with binary sessions, not multiparty as in here, but the main arguments presented therein remain valid also in the multiparty case.

```
8     ok( void );
9       RP -> U: ok( void ),
10    fail(string);
11      RP -> U: fail( string )
12    }
13  }
14
15  public publicOpenID: OpenID
16
17  main
18  {
19    rp[RP], u[U] start ip[IP]: publicOpenID( k );
20
21    ask@u( "[u] Insert Username", user );
22
23    u.user -> rp.user: username( k );
24    rp.user -> ip.username: username( k );
25
26    ask@u( "[u] Insert Password", pwd );
27
28    u.pwd -> ip.password: password( k );
29
30    ask@ip(
31      "[ip] Accept username '" + username +
32      "' and password '" + password + "' ?",
33      accept
34    );
35    if (accept == "yes")@ip {
36      ip -> rp: ok( k );
37      rp -> u: ok( k );
38      show@u( "[u] Authentication successful!" )
39    } else {
40      ip."Invalid credentials" -> rp.error: fail( k );
41      rp.error -> u.error: fail( k );
42      show@u( "[u] " + error )
43    }
44  }
```

In the code above, we start by declaring a protocol `OpenID` where a user `U` wants to authenticate to a relying party `RP` using an external identity provider `IP`. The user starts by sending her username to `RP`, which forwards it to `IP`; then, the user sends her password to `IP`. In Lines 7–12, the identity provider `IP` selects either branch `ok` or `fail` on the relying party, which terminates the protocol by forwarding the choice to the user.

We implement the protocol in our choreography with the session `k` and the three processes `rp`, `u`, and `ip`. Lines 21–28 implement Lines 4–6 in the protocol, by asking information to the user and then communicating it. In Line 30, the `ip` asks whether the

presented username/password credentials are valid; in Lines 35–43, the choice is then forwarded to processes `rp` and `u`, completing the protocol.

1.4.2 Session Interleaving

In the example below, we give an implementation of a system where two sessions are interleaved to reach a common goal.

```
1  program buyerseller;
2
3  protocol BuyerSeller {
4    Buyer -> Seller: buy( string );
5    Seller -> Buyer: price( int );
6    Buyer -> Seller: {
7      ok(void),
8      abort(void)
9    }
10 }
11
12 protocol AskUser {
13   Application -> User: question( string );
14   User -> Application: {
15     yes( void ),
16     no( void ),
17     maybe( void )
18   }
19 }
20
21 public a : BuyerSeller
22 public b : AskUser
23
24 main
25 {
26   buyer[Buyer] start seller[Seller] : a( k );
27   buyer."Coffee" -> seller.product : buy( k );
28   seller.100 -> buyer.price : price( k );
29
30   buyer[Application] start user[User]: b( k2 );
31   buyer.( "Can I pay " + price + "?" )
32     -> user.question: question( k2 );
33
34   local@user( s = question );
35   ask@user( s, answer );
36
37   if (answer == "yes")@user {
38     user -> buyer: yes( k2 );
39     buyer -> seller: ok( k );
```

```
40      show@seller( "[seller] Product sent!" )
41    } else {
42      user -> buyer: no( k2 );
43      buyer -> seller: abort( k );
44      show@seller( "[seller] Transaction aborted!" )
45    }
46  }
```

Above, we specify two protocols: `BuyerSeller` and `AskUser`. In protocol `BuyerSeller`, a `Buyer` asks a `Seller` for the price of a product using operation `buy`; the `Seller` replies with operation `price`, and then the `Buyer` finally notifies the `Seller` of whether she wishes to proceed with the purchase (operation `ok`) or not (operation `abort`). Protocol `AskUser` is very simple: an `Application` asks a `User` a question and then waits for an answer from the `User`, which can be `yes`, `no`, or `maybe`.

The choreography starts by instantiating protocol `BuyerSeller` as session `k`, on which the buyer asks for some coffee and the seller replies with the price 100. Now, in Line 30, the buyer starts another session `k2` with another process `user` by playing role `Application`. The buyer uses session `k2` to asks whether the user is willing to pay the price required by the seller. Depending on the user's choice, buyer completes session `k` accordingly (Lines 37–45).

Observe that in session `k2` we are not implementing option `maybe`, since we do not need it. Our type checker will still accept the choreography as safe, following our typing discipline from § **??** (see § **??** for details).

1.4.3 Service Selection and Delegation

We report an example from the Ocean Observatories Initiative (OOI) [11], in which a user connects to an instrument for reading environmental data through a service responsible for authorising user commands.

```
1  program instrument;
2
3  protocol AuthCommand {
4    U -> A: username( string );
5    U -> A: password( string );
6    A -> U: {
7    valid(Connect@U),
8    fail(void)
9    }
10 }
11
12 protocol Connect {
13   U -> R: connect( string );
14   R -> U: {
15     ok( ExecCommand@C ),
16     unavailable( void )
17   }
```

```
18 }
19
20 protocol ExecCommand {
21   C -> I: {
22   readTemperature(void);
23     I -> C: result(string),
24   readPressure(void);
25     I -> C: result(string)
26   }
27 }
28
29 public a : AuthCommand
30 public b : Connect
31 public instrument1 : ExecCommand
32 public instrument2 : ExecCommand
33
34 define findAndExec( u, r )( connect[Connect: u[U], r[R]] )
35 {
36   ask@u( "[u] What instrument
37     do you want to connect to? (inst1/inst2)",
38     name );
39   u.name -> r.name: connect( connect );
40   if ( name == "inst1" )@r {
41     r[C] start i[I]: instrument1( exec );
42     r -> u: ok( connect( exec ) );
43     u -> i: readTemperature( exec );
44     i."28 C" -> u.temp: result( exec )
45   } else if ( name == "inst2" )@r {
46     r[C] start i[I]: instrument2( exec );
47     r -> u: ok( connect( exec ) );
48     u -> i: readTemperature( exec );
49     i."2 C" -> u.temp: result( exec )
50   } else {
51     r -> u: unavailable( connect )
52   }
53 }
54
55 main
56 {
57   u[U] start a[A]: a( auth );
58   ask@u( "[u] Insert username", username );
59   ask@u( "[u] Insert password", pwd );
60   u.username -> a.username: username( auth );
61   u.pwd -> a.pwd: password( auth );
62   ask@a( "[a] Confirm credentials? (yes/no) : "
63     + username + " : " + password,
```

```
64      confirm );
65    if ( confirm == "yes" )@a {
66      a[U] start r[R]: b( connect );
67      a -> u: valid( auth( connect ) );
68      findAndExec( u, r )( connect )
69    } else {
70      a -> u: fail( auth )
71    }
72  }
```

We have three protocols. In protocol `AuthCommand`, a user `U` sends her credentials to an authoriser `A` which decides whether the credentials are valid or not. If the credentials are valid, then the authoriser delegates to the user access to a session for connecting to an instrument. The latter will follow protocol `Connect`, in which the user asks a registry `R` for a connection to a specific instrument. If the instrument is available, the registry delegates to the user a session for communicating with the instrument she requested, which follows protocol `ExecCommand`. Finally, in protocol `ExecCommand`, a client `C` (which will be the user in our choreography) asks an instrument `I` for a temperature or a pressure reading.

In the choreography, Lines 57–72 implement protocol `AuthCommand`. If successful, the authoriser starts a session `connect` with the registry and delegates it to the user; the choreography then proceeds by calling procedure `findAndExec`. In the procedure, the user selects an instrument and asks the registry for being connected to it. If the instrument is known by the registry, the latter creates a session with the instrument and then delegates it to the user. Finally, the user uses the instrument for getting a reading.

1.4.4 Recursive Protocols

Our last example is about recursive protocols. Specifically, we define a protocol for streaming some packets from a server to a client.

```
1  program stream;
2
3  protocol StreamingProtocol {
4    S -> C: packet( string );
5    S -> C: {
6      again( void ); StreamingProtocol,
7      stop( void )
8    }
9  }
10
11 public a : StreamingProtocol
12
13 define doStreaming
14      ( c, s )
15      ( stream[StreamingProtocol: c[C], s[S]] )
16 {
```

```
17    ask@s( "[s] Input data to send for packet number " + i,
18      data );
19
20    s.data -> c.data: packet( stream );
21    local@c( result = result + data );
22    local@s( i = i + 1 );
23    if ( i < nPackets )@s {
24      s -> c: again( stream );
25      doStreaming( c, s )( stream )
26    } else {
27      s -> c: stop( stream );
28      show@c( "[c] Received data: " + result )
29    }
30  }
31
32  main
33  {
34    c[C] start s[S]: a( k );
35    ask@s( "[s] How many packets do you want to send?",
36      nPackets );
37    local@s( i = 0 );
38
39    doStreaming( c, s )( k )
40  }
```

Above, we have only one protocol `StreamingProtocol`. In the protocol, the server `S` sends a packet to the client `C` and then informs the client on whether there are more packets or not. In the first case (operation `again`), the protocol continues; otherwise (operation `stop`), it terminates immediately.

In the choreography, a client process `c` starts a server process `s` to implement the protocol on session `k`. The server then establishes how many packets to send and the choreography proceeds by calling procedure `doStreaming`. In the procedure, the server gets some data to send to the client and sends it. The client aggregates the received data in its local variable `result`. Thereafter, the server checks whether there are more packets to send. If so, the server informs the client of continuing and the procedure recurs by invoking itself; otherwise, the server informs the client that there are no more packets and the choreography terminates.

1.5 Related Work

From a language perspective, Chor is essentially an implementation of the Choreography Calculus presented in Chapter **??**, so we refer the reader to § **??** for a discussion on related work about the language design.

On the side of tools for choreographic programming, the works nearest to Chor are WS-CDL [13] and BPMN [1]. The main difference is that Chor is strongly typed and supports the type checking of sessions against formally-defined protocols, whereas WS-CDL

and BPMN do not come with this kind of typing discipline and are thus more error-prone. Moreover, the semantics of both languages is informally specified, making their interpretation potentially unclear, whereas Chor follows the formal semantics given in Chapter **??**. Consequently, it is not possible to formally reason about safety properties on WS-CDL and BPMN, whereas the design of Chor is based on the foundations investigated in Chapter **??**. However, regarding EPP, Chor targets Jolie code instead of the endpoint model given for the Choreography Calculus in § **??**. An interesting future work is therefore to formally prove that adopting the Jolie model given in Chapter **??** for endpoint programs does not alter the safety properties guaranteed in § **??**.

1.6 Conclusions

We presented Chor, a prototype framework for Choreographic Programming. Through examples, we evaluated Chor against a series of use cases and shown that it is already expressive enough to handle, e.g., the interleaving of protocols, service selection, and session delegation.

1.6.1 Future Work

We discuss some directions for future work on Chor.

Language Extensions. Chor is an implementation of the model in Chapter **??**, the Choreography Calculus, which does not support all the features presented in this dissertation in the later Chapters. We plan to extend Chor to handle the composition of choreographies and round-trip development as formalised, respectively, in Chapters **??** and **??**.

Global Deployment. Chor projects Jolie programs with a default deployment configuration that can be edited afterwards for each endpoint. This may be inconvenient for the programmer, since a choreography may describe many participants. Therefore, we plan to introduce a global deployment language for choreographies, from which the deployment configuration of each endpoint could be automatically generated. We envision that this extension could be relevant for facilitating checks on the correctness of a deployment configuration of a system, e.g., consistent I/O connections.

Scaffolding. Since in our model a protocol and a session behaviour in a choreography have similar structures, we could implement a scaffolding tool in our IDE that, given a protocol, would generate a prototype choreography with "dummy" data that implements it. Then, programmers would refine and interleave different prototypes to obtain the desired behaviour.

Local Code. The local actions in Chor are still very limited and are only meant to exemplify the idea of introducing internal computation at endpoints in choreographies. We plan to investigate how to embed an entire language for internal actions, such as Jolie or Java, so to reach a more general result. This would also be helpful for reusing already existing libraries from other frameworks in choreographies.

Bibliography

[1] Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/. (Cited on page 14.)

[2] SOAP Specifications. http://www.w3.org/TR/soap/. (Cited on page 6.)

[3] Web Services Addressing. http://www.w3.org/TR/ws-addr-core/. (Cited on page 7.)

[4] Chor. Programming Language. http://www.chor-lang.org/. (Cited on pages 1 and 8.)

[5] Eclipse. The Eclipse IDE. http://www.eclipse.org/. (Cited on pages 1 and 5.)

[6] J. N. Gray. *Notes on data base operating systems*. Springer, 1978. (Cited on page 7.)

[7] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, volume 43(1), pages 273–284. ACM, 2008. (Cited on page 1.)

[8] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541, 2008. (Cited on page 8.)

[9] F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *ECOWS*, pages 13–22, 2007. (Cited on page 2.)

[10] OASIS. Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. (Cited on page 2.)

[11] OOI. Ocean Observatories Initiative. http://www.oceanobservatories.org. (Cited on pages 2 and 11.)

[12] OpenID. Specifications. http://openid.net/developers/specs/. (Cited on page 8.)

[13] W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/, 2004. (Cited on page 14.)

[14] Xtext. The Xtext Framework. http://www.eclipse.org/Xtext/. (Cited on page 5.)