Verified Parameterized Choreographies*

Robert Rubbens(✉) [0000−0002−5638−5945], Petra van den Bos [0000−0002−9212−1525], and Marieke Huisman [0000−0003−4467−072X]

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{r.b.rubbens,p.vandenbos,m.huisman}@utwente.nl

Abstract. Choreographies are useful for modelling systems with multiple simultaneously executing and communicating participants, e.g. distributed systems. VeyMont can verify correctness of choreographies and generate verifiably correct code that implements the choreography. Initially, it supported only fixed sets of participants. However, realistic systems are often *parameterized*: they scale according to some parameter N. This paper extends VeyMont with *parameterized choreographies*, making VeyMont more usable for realistic case studies. Specifically, we add parameterized primitives such as *participant families* and *parameterized communication*. We encode these primitives using a structured parallelism primitive from the underlying verifier VerCors, and by using conditionals in the endpoint projection, partially delaying projection until run time. We illustrate the encoding with a *distributed summation* choreography, and prove it correct with VerCors.

Keywords: Choreographies · Parameterization · Deductive verification

1 Introduction

Distributed systems are not just ubiquitous, they are indispensable for networked systems on a global scale. Unfortunately, guaranteeing robustness of distributed systems is still a challenge. Consider a participant of a distributed system, waiting for a message that will never be sent. Clearly, this system cannot function reliably. This type of bug is called a *communication deadlock*, and ideally a distributed system would be free of deadlocks. Another aspect of robustness is that of functional correctness: maybe the distributed system never deadlocks, but does the system actually compute the *correct* result?

An approach to improve the reliable development of distributed systems is to use the *top-down* formalism of *choreographies* [15]. In its purest form, a choreography is a series of message exchanges between participants, called *endpoints*. Choreographies have two primary properties [15]. The first is communication deadlock freedom: no endpoint will be stuck waiting for a message that will never be sent. The second is *message fidelity*: an endpoint will never receive a message of a different type than it is expecting. Choreographies also support the *endpoint projection*, which generates an implementation for a given endpoint.

* Supported by the NWO VICI 639.023.710 Mercedes project.

```
1 choreography summation2() {
2   endpoint a = Node(int());
3   endpoint b = Node(int());
4   run {
5     communicate a.sum -> b.in;
6     communicate b.sum -> a.in;
7     a.update(); b.update(); } }
```

(a) For two endpoints `a` and `b`

```
1 choreography summationN(int N) {
2   endpoints nodes[i := 0..N] = Node(i, int());
3   run {
4     while ((\endpoints nodes[i := 0..N]; nodes[i].n < N-1)) {
5       communicate nodes[i := 0..N-1].sum -> nodes[i+1].in;
6       communicate nodes[N-1].sum -> nodes[0].in;
7       nodes[i := 0..N].update(); } } }
```

(b) Parameterized for `N` endpoints

Fig. 1: Distributed summation choreographies

Figure 1a shows an example of a choreography that sums the values of two endpoints. Lines 2 and 3 declare the endpoints `a` and `b` of type `Node`, and initialize their `sum` fields with a random integer. On lines 5 and 6 each endpoint sends their local sum to the `in` field of the other. On line 7, they update their `sum` fields with the sum of their initial value and the value of the `in` field. The `sum` field of each endpoint now contains the sum of both the initial values. As an example in Java-like syntax, the endpoint projection of Fig. 1a for endpoint `b` is:

```
b.in = chan_ab.readValue(); chan_ba.writeValue(b.sum); b.update();
```

To verify choreographies like Fig. 1a, VeyMont was developed, a verifier and code generator for choreographies [2]. It supports functional correctness verification of choreographies with contract annotations, such as pre- and postconditions and asserts. When generating code with VeyMont, verification annotations are preserved [19], which means correctness of the generated code can be established independently from the initial choreography. This allows safe modification of generated code. VeyMont is built on top of VerCors, a deductive verifier for concurrent and parallel software. Besides languages such as Java and C, VerCors also supports the internal Prototypical Verification Language (PVL), a Java-like language intended for rapid prototyping of verification features.

To verify a choreography, VeyMont applies the *choreographic projection*, which transforms a choreography into a PVL program that combines the behaviour of *all endpoints* into a *single program* [2]. This is in contrast to the endpoint projection, which slices a choreography in such a way that only the parts relevant for one specific endpoint remains. The choreographic projection has two goals: 1) to make the choreography verifiable with an off-the-shelf program verifier like VerCors, and 2) to add annotations for correctness aspects such as deadlock freedom and memory safety. The first goal is achieved by modelling communication with regular assignment, and preserving composite statements such as `if` and `while`. To illustrate, the choreographic projection for Fig. 1a is:

```
b.in = a.sum; a.in = b.sum; a.update(); b.update();
```

The second goal is achieved by encoding correctness aspects into PVL [2,19]. Then, if the projection is verified, the choreography respects its contracts [11].

However, verifying regular choreographies is not enough. Instead, realistic case studies often scale with some parameter N, and hence require *parameterized choreographies*. In Fig. 1b a distributed sum choreography is parameterized

by N (line 1). Instead of defining endpoints individually, line 2 defines an *endpoint family*, which is a range of endpoints, its size determined by a symbolic expression. The choreography in Fig. 1b generalises the approach from Fig. 1a: for $N-1$ rounds, each node will send its partial sum to a neighbouring node, as done on lines 5 and 6. Then, each node will update its partial sum (line 7), after which the while loop will repeat. When the while loop terminates, each node will know the sum of all initial values.

Parameterization of *both* the choreographic and endpoint projection is still an open problem. To enable *verification* and *code generation* for choreographies like Fig. 1b, this paper discusses how to extend VeyMont with parameterization.

Contributions We define choreographies with parameterization by adding parameterized primitives to the choreographic language of VeyMont (Section 3). In particular, we add *endpoint families*, which are ranges of endpoints with their size defined by a symbolic expression. We also add a *parameterized communication statement*, which communicates a message according to a user-defined one-to-one mapping between two possibly overlapping endpoint family ranges.

To verify parameterized choreographies, we extend the choreographic projection to use a structured parallelism primitive, the `par` block, to encode the semantics of the parameterized communication statement (Section 4). We identify a fragment of the choreographic language for which memory safety annotations can be automatically generated, preserving full automation of the verification process. E.g. we limit the syntax of the parameterized communication statement such that it may only access memory in a certain pattern. Other parameterized syntax, such as endpoint families and parameterized expressions, can be encoded using mathematical sequences and universal quantifiers. We also extend pre-existing VeyMont features (deadlock freedom and shared memory [2,19]) to support parameterization.

To support parameterization, the endpoint projection must generate an implementation for an unknown but fixed amount of endpoints (Section 5). This is important, because in a parameterized choreography, the sizes of endpoint families are defined by symbolic expressions. We make part of the endpoint projection conditional on the runtime value of endpoint family indexing expressions, effectively delaying projection until *run-time*. That way, instead of generating one implementation *per* endpoint, we generate an implementation for a *range* of endpoints, which behaves based on the run-time value of the current endpoint index. This encoding provides executable code with preserved verification annotations, allowing deductive verification separate of the choreography.

We illustrate the choreographic and endpoint projection with the distributed sum running example, which we have proven correct with VerCors in the artefact [17]. Finally, we discuss related work (Section 6).

2 PVL

The Prototypical Verification Language (PVL) is an object-oriented programming (OOP) language with contracts and assertions. It is used to prototype

x, y, z ::= field, v, u, w ::= variable, m ::= method

C ::= class name

T ::= `int` | `boolean` | `seq<`T`>` | C | $\cdots$

prog ::= $\overline{\textbf{decl}}$ **decl** ::= *class definition* | **chor**

E ::= v | r | F | $\cdots$

H ::= v | r | F | $\cdots$ | F`[`H`]` | H`.`x | `this`

R ::= H | `Perm(`H`.`x`,` H`)` | R `**` R | H `==>` R

K ::= `requires` R`;` `ensures` R`;`

S ::= K `par (`T v `=` H `..` H`)` S | $\cdots$

(a) Abbreviated OOP fragment

```
1  class Role {
2    int x; // Field declaration
3    requires Perm(this.x, 1);
4    ensures Perm(this.x, 1) **
5      this.x == \old(this.x) + 1;
6    void incrX() {
7      x += 1; } }
```

(b) `Role` class definition example

e, a, b ::= endpoint F, G ::= endpoint family,

chor ::= K `choreography(`$\overline{T\ v}$`)` `{` $\overline{D_{\texttt{chor}}}$ `}`

D_{chor} ::= `endpoint` e `=` $C(\overline{H})$`;`

| `endpoint` F`[`v `:=` 0 `..` H`]` `=` $C(\overline{E})$`;`

| K `run {` $S_{\texttt{chor}}$ `}`

r, p ::= e | F`[`E`]` α, β ::= r | F`[`v `:=` E `..` E`]`

S_{chor} ::= `if (`H_{chor}`)` S_{chor} S_{chor} | `assert` R_{chor}`;`

| `loop_invariant` R_{chor}`;` `while (`H_{chor}`)` S_{chor}

| `endpoint` α`:` S_{ep}

| `channel_invariant` R_{chan}`;` `communicate` α`:` H `->` α`:` H`;`

S_{ep} ::= H`.`$m(\overline{H})$`;` | H `:=` H`;`

H_{chor} ::= `(\endpoint` α`;` H`)` | H_{chor} `&&` H_{chor}

R_{chor} ::= `(\endpoint` α`;` R`)` | R_{chor} `**` R_{chor} | H_{chor}

R_{chan} ::= Inline R | `\msg` | `\sender` | `\receiver`

(c) Choreographic fragment

Fig. 2: Formal syntax of PVL

verification features in VerCors. We use it to formalize a semantics for choreographies by defining transformations from choreographies to PVL.

The syntax of PVL is shown in Fig. 2a; this description follows the order in the figure. We first define names for several types of declarations, including some typical elements for each declaration type. E.g. the names x, y and z will always refer to a field of a class. PVL supports several built-in types, such as integers, booleans, and sequences. Each class C is also a distinct type. A PVL program consists of zero or more classes or choreographies. The syntax of classes C and statements S is similar to Java, except for the `par` block statement, discussed later. A more complete syntax can be found in [18]. Here, we show an example in Fig. 2b. The choreography syntax from Fig. 2c is discussed in Section 3.

Figure 2a defines several kinds of expressions. Pure expressions E only use local variables, endpoints and immutable value constructors (e.g. sequences). Endpoints, written as r, are choreography participants, and further explained in Section 3. A heap-dependent expression H extends E with field dereferencing.

A resource expression R is a heap-dependent expression, a permission expression using `Perm`, a resource-combining operator `**` or a conditional resource using the implication operator `==>`. A permission consists of a field access expression together with a fraction between 0 and 1 inclusive, where 1 indicates a read-write permission, and a fraction between 0 and 1 exclusive a read-only permission. The separating conjunction (`**`) can compose resource expressions

when the sum of fractions per field do not exceed 1. E.g. writing `Perm(o.x, 1)` is identical to `Perm(o.x, 1/2) ** Perm(o.x, 1/2)`. For boolean expressions, `**` behaves as `&&`. We define the *footprint* of an expression or statement as the permissions required to evaluate or execute it. E.g. for some positive fraction f, a possible footprint of `o.x` would be `Perm(o.x, f)`. The statement `o.x = o.y` would have footprint `Perm(o.x, 1) ** Perm(o.y, f)`.

Two statements manipulate permissions directly. The `inhale` R statement adds the permissions R to the current thread. The `exhale` R statement first checks if the current thread actually has all the permissions R, and then removes them from the current thread. Verification fails if the permissions are not available. These statements are verification primitives for encoding programming language semantics. E.g. acquiring a lock that guards write permission to the field `o.x` can be modelled with the statement `inhale Perm(o.x, 1)`. When used with plain boolean expressions, `inhale`/`exhale` behave as `assume`/`assert`.

The kinds of expressions E, H and R distinguish expression capabilities. Specifically, expressions E will only refer to local variables and endpoints, H can mention heap locations, and R can mention permissions. VerCors then uses annotations in the form of R to check memory safety of expressions H.

Contracts K are written using the `requires` and `ensures` keywords, and can be added to methods, choreographies and `par` blocks. The `par` block is an important concurrency primitive in PVL [6]. The first part of the `par` block is the contract, which specifies behaviour from the perspective of one thread. Then follows a binder that will contain the index of each thread, followed by a range that determines the number of threads. The body of a `par` block consists of standard imperative statements, such as `if`, `while`, etc. The semantics of a `par` is as follows: when reaching a `par` block, N subthreads execute the `par` block body in parallel. The parent thread waits until the subthreads finish.

3 Choreographies

We will next define the syntax for parameterized choreographies, as well as give an intuition for the semantics (Section 3.1). The parameterization extension also requires integration with existing *deadlock freedom* [2] and *shared memory* [19] support of VeyMont (Sections 3.2 and 3.3). Finally, we introduce the full distributed summation running example (Section 3.4).

3.1 Syntax

The syntax for choreographies in VeyMont is shown in Fig. 2c; again, the description in this section follows the order in this figure. We also informally describe its semantics. Sections 4 and 5 describe the actual semantics of choreographies by defining two transformations into the OOP fragment of PVL.

Declarations A choreography consists of a contract, a series of arguments, and a series of choreographic declarations, which can be an endpoint, an endpoint

family, or a `run` declaration. A single endpoint has a name (e, a, or b), a class type C and an argument list. The argument list is passed to the constructor at run-time, which creates an instance of C to represent the endpoint.

A parameterized endpoint is an *endpoint family*, which additionally has a size parameter. Note that this parameter can be *symbolic*. Therefore, to verify or generate code for parameterized choreographies, one must either use e.g. an SMT solver that can do so symbolically, or somehow delay inspection of this size until run-time, when the symbolic parameter is instantiated. The parameter is also allowed to depend on the heap. This is because the parameter is evaluated before the endpoint family is initialized. When each member of an endpoint family is constructed at run-time, the binder v is also in scope, which contains the index of the current endpoint within the endpoint family. The endpoint family is represented at run-time as a sequence of instances of C. An example of a parameterized endpoint is shown in Fig. 1b on line 2.

The `run` declaration has a contract and a series of choreographic statements. Essentially, each endpoint executes the `run` declaration by only executing the choreographic statements related to the endpoint. The contract of the `run` declaration differs from the choreography contract as follows: the choreography precondition holds before endpoint initialization, the `run` precondition holds after initialization and before `run` is executed. Conversely, the `run` postcondition holds when an endpoint finishes, the choreography postcondition holds after *all* endpoints have finished. Examples of `run` declarations are in Figs. 1 and 3.

Endpoint references There are two notations for endpoint references: r and α. The notation r refers to a particular endpoint, which can be be either a singular endpoint e, or a family F indexed by a pure expression. E.g. F`[N-1]` selects the last endpoint of a family F of size N. The notation α extends r with ranges of endpoint families as follows: F`[`i `:=` E_l `..` E_h`]`, for a family F, a binder i, and a half-open range $[E_l, E_h)$. E.g. F`[tid := N/2..N]` refers to all endpoints of F with index $\in [\frac{N}{2}, N)$. In this case, the family has to have at least size N. The binder i is also used in endpoint expressions, explained later. This notation is inspired by related work of Ng et al. [16], as discussed in Section 6.

Indexing into endpoint families using α is only allowed with pure expressions E. This is deliberate, as it ensures any indexing operation is heap independent, meaning no permission annotations are necessary for endpoint family indexing. This also ensures any indexing operation can be executed by any endpoint of a choreography. This allows evaluating indexing expressions in any endpoint context, which is important for communication statements, discussed later.

Statements Branches, asserts and loops are *choreographically transparent*: they are only executed by the endpoints that occur within them. If an endpoint is not mentioned in a choreographically transparent statement, the endpoint skips it.

An endpoint statement is a local action of an endpoint. It requires an α, meaning the action can apply to a singular endpoint or a family range. We allow method calls and assignments on endpoints. For example, in the statement "`endpoint` e`:` e`.`m`()`" the method call e`.`m`()` will be executed by endpoint e.

The formal syntax for endpoint statements is slightly more general than what the choreographic and endpoint projection can handle. For example, parameterized endpoint statements can only do method calls directly on endpoints. We enforce these restrictions syntactically, and further discuss them in Section 4.

A `communicate` statement specifies communication. Through the α notation, communication can either be between two singular endpoints, or between two family ranges with an injective mapping. Injectivity is ensured by both the choreographic and endpoint projection through explicit checks. Communication statements consists of a channel invariant, a sending endpoint, the message to be sent, the receiving endpoint, and the destination in which the message will be received. The channel invariant specifies a property over the message, i.e. an invariant over values in the channel. The endpoint projection (Section 5) creates a channel for each `communicate` [19,2]. The primitives `\sender`, `\receiver` and `\msg` may be used to refer to the sender, receiver, and message respectively in the channel invariant. For example, `channel_invariant \msg > 2` specifies all messages sent over the channel must be bigger than 2. Examples of singular and parameterized `communicate` statements are shown in Fig. 1.

For both endpoint and communication statements, whenever the endpoint annotations are obvious, they are omitted. E.g. in `communicate a.x -> b.y`, we omit the `a:` and `b:` annotations.

Expressions We distinguish two kinds of choreographic expressions. The first expression type is H_{chor}, which is essentially a list of endpoint expressions, composed using `&&`. An endpoint expression is an expression tagged with the endpoint that should evaluate it. If the endpoint expression introduces a binder, this binder can appear inside the tagged expression. E.g. consider this expression:

$$\texttt{(\textbackslash endpoint } F\texttt{[}i\texttt{ := 0 .. N]; } F\texttt{[}i\texttt{].x == }i\texttt{)}$$

This states that endpoints in family F have a field `x` equal to the endpoint index. In addition, when an endpoint evaluates the endpoint expression, it must do so using only its own memory. This is further explained in Section 3.3. The second expression type is R_{chor}, which is similar to H_{chor} except that R_{chor} can also introduce permissions, and compose them with the separating conjunction `**`.

3.2 Deadlock freedom

Deadlocks occur if an endpoint is waiting for a message that will never be sent. This can happen when branches are involved. Consider the choreography `communicate a.x -> b.y; ...`, which is just a series of communications and local actions. Because of the simple structure of this choreography, each send is guaranteed to be paired with a receive, and hence it cannot deadlock. Now consider the choreography `if (a.x && b.x) communicate a.y -> b.y`, which consists of one branch and one communicate. Here, a deadlock is possible: when `!a.x && b.x` holds, `b` will enter the body of the `if` statement, and `a` will skip it. This is because of the semantics of choreographic expressions, where `a` will only execute expressions relevant to `a`, and vice versa for `b`. This will result in `b` waiting for a message, even though it will never be sent, because `a` skipped it.

To prevent deadlocks, choreographies need branch unanimity [2]. A branch is unanimous if the condition of a branch evaluates to the same value for each participating endpoint. In other words, all endpoints have to agree "unanimously" on the condition of the branch. VeyMont checks this automatically [2]. Taking the previous example, the branch is unanimous if `a.x == b.x` always holds before the `if`. Branch unanimity also applies to `while` loops, and hence ensures that either both endpoints encounter the communicate statement, or they both skip it, and not something in between. For a finite set of endpoints, a verification condition for branch unanimity is straightforward to generate, following the example above. We extend branch unanimity for endpoint families in Section 4.5.

3.3 Shared memory

Supporting shared memory in choreographies requires memory safety annotations. For example, the statement `endpoint a: b.x := y` (`a` assigns `y` to `b.x`) is safe *if and only if* endpoint `a` has write permission for `b.x`.

VeyMont supports *endpoint ownership annotations* to bind a permission to an endpoint [19]. E.g. `(\endpoint a; Perm(b.x, 1))` states that `a` needs write permission for `b.x`. VeyMont checks if endpoints in a choreography do not use memory that they do not own by transforming the endpoint ownership annotations into PVL permissions. This is done as part of the choreographic projection [19]. Summarizing, the permission from the previous example would be transformed into `Perm(b.x, 1, a)`, using a special encoding to put the endpoint owner as metadata in the permission. The transformation also guards all field accesses as follows. First, the transformation generates the function `read_f`:

```
requires Perm(o.f, 1, e); int read_f(endpoint e, object o) = o.f;
```

The precondition of `read_f` forces permission with proper metadata to be available at every field access. The transformation also replaces every field access with an invocation of `read_f`. E.g. the expression `(\endpoint a; b.f > 0)` is transformed into `read_f(a, b) > 0`.

The shared memory support of VeyMont is straightforward to integrate with parameterization, because parameterized permissions can be encoded using universal quantifiers. This is sufficient, as quantifiers are natively supported by the underlying verifier VerCors. The only integration required is for the choreographic projection to explicitly apply the shared memory encoding. This is done by using the *confined memory mode* operator, introduced in Section 4.

3.4 Running example: distributed summation

We now discuss a choreographic formulation of the ring-based distributed summation algorithm. It will also be used to illustrate the choreographic projections.

Algorithm encoding In the choreography in Fig. 3, each endpoint only communicates with its two neighbouring endpoints (resp. predecessor and successor), simulating a ring topology of size N. An endpoint initially knows only its own

```
requires N >= 2;
choreography summation(int N) {
  endpoints ns[i := 0..N] = Node(i, int());
  requires (\endpoints ns[i := 0..N]; ns[i].sum == ns[i].v);
  ensures (\endpoints ns[i := 0..N]; ns[i].sum == sum(ns, 0, ns[i].n));
  run {
    invariant (\endpoints ns[i := 0..N]; ns[i].total == sum(ns, ns[i].i, ns[i].n));
    while ((\endpoints ns[i := 0..N]; ns[i].n < N-1)) {
      channel_invariant \msg == sum(ns, \sender.i, \sender.n);
      communicate ns[i := 0..N-1].sum -> ns[i+1].in;
      channel_invariant \msg == sum(ns, \sender.i, \sender.n);
      communicate ns[N-1].sum -> ns[0].in;
      ns[i := 0..N].update(); } } }
```

Fig. 3: Distributed summation choreography

value. The goal is that each endpoint eventually knows the sum of the values of all endpoints. The sum is calculated using the `sum` function, with parameters: a sequence of nodes, the starting index and the number of indices to include in the sum. At each iteration, the endpoints send the current partial total to their successor. Then, they receive a partial total from their predecessor, and add their own local value to it. This yields a new total. After looping $N-1$ times, each endpoint knows the network total.

A key difference between the algorithm and the choreographic encoding is how the network structure is encoded. Instead of using the modulo operator, we apply an insight from previous work and "linearize" the ring communication into two separate communications [16,7]. One is parameterized over the range 0 to $N-1$, and the other communication is from $N-1$ to 0, closing the loop. This shows that circular topologies can be encoded using simpler linear structures.

Verification outline The choreography contains several verification annotations, which give an outline of the correctness proof. Essentially, as partial sums are communicated between endpoints, the partial sum of each endpoint converges towards the true total in $N-1$ iterations.

Permission annotations, ghost state and proof steps necessary to verify Fig. 3 are omitted for ease of presentation. In particular, verification requires a lemma that uses the symmetry of addition to show that a sum starting at endpoint `ns[`i`]` equals the sum starting at endpoint `ns[`j`]`. The version with full verification annotations is available in the artefact [17]. In particular, we verified that each endpoint computes the same network total.

4 Choreographic Projection

We will now discuss the choreographic projection operator $\{\!|\bullet|\!\}$. Its purpose is to encode the choreography into a PVL program that VerCors can verify. This way, if the PVL program verifies, the choreography is correct. Otherwise, there might be a bug, either in the choreography, or in its specification.

```
communicate a: a.balance > a.n -> b: b.ok;

if (a.balance > a.n && b.ok) {
  communicate a.n -> b.n
  a: a.balance := a.balance - a.n;
  b: b.balance := b.balance + b.n; }
```

(a) Input choreography.

```
b.ok = a.balance > a.n;
assert a.balance > a.n == b.ok;
if (a.balance > a.n && b.ok) {
  b.n = a.n;
  a.balance = a.balance - a.n;
  b.balance = b.balance + b.n; }
```

(b) Output PVL.

Fig. 4: Encoding of a choreography that models a bank transfer.

To encode parameterized constructs, we use two primitives from the underlying verifier VerCors: the `par` block for structured parallelism and universal quantifiers. We impose restrictions on the allowed syntax to ensure the required annotations can be generated automatically. For non-parameterized constructs, the encoding transforms all `communicate` statements into plain assignments, and keeps other primitives, i.e. `if`, `while`, assignments and method calls. Effectively, the choreographic projection picks a representative interleaving of all possible interleavings of choreography, and then encodes it in plain PVL. This is sound, as each endpoint is also verified to be memory safe. Memory safety guarantees non-interference, and therefore the behaviours of all interleavings are equivalent.

The choreographic projection operator has two modes. The plain mode, written as $\{\!|\bullet|\!\}$, encodes the choreography as a sequential object-oriented PVL program, while adding additional checks for deadlocks. The confined memory mode, written as $\{\!|\bullet|\!\}_r$, ensures the argument is encoded such that only memory of endpoint r is used (see Section 3.3). The confined memory mode is used by the plain mode when an endpoint context annotation occurs.

We only list the rules that are key to transforming the examples. The full listing can be found in [18]. Note that the choreographic projection results in a program intended for *verification*. The resulting program is an abstracted version of the choreography, which behaves as if all endpoints are sharing one thread, yet does not exclude any concurrent behaviours. Transformation for the purpose of execution is done by the endpoint projection explained in Section 5.

4.1 Non-parameterized example

To give an intuition for the choreographic projection, we first show a concrete example. Figure 4a shows the input choreographic code. Figure 4b shows the output PVL code, created using the choreographic projection $\{\!|\bullet|\!\}$. There is a close correspondence between the left and the right listing: each statement is encoded using the corresponding rule from Fig. 5. The only new statement is the `assert`, added by rule CpIf, which checks deadlock freedom (see Section 3.2).

4.2 Non-parameterized projection rules

We will now discuss the rules required to transform the example in Fig. 4, shown in Fig. 5. Rule CpAssign pattern matches on the subparts of the choreographic

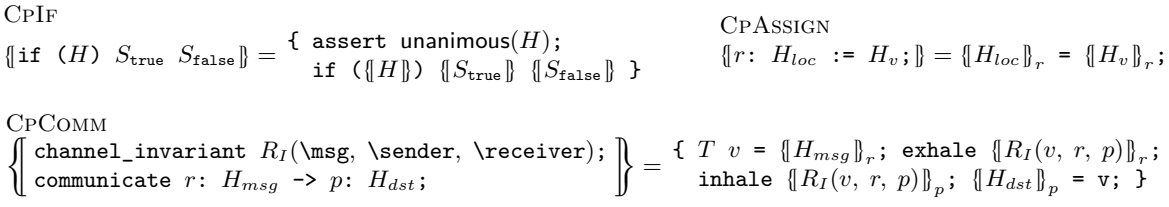

Fig. 5: Non-parameterized choreographic projection rules

assignment statement on the left side of =, and shows how to construct the projected statement on the right. In this case, the choreographic projection is applied to H_{loc} and H_v, removing the endpoint labels from the resulting statement. Within rules, subscripts such as *loc* and v are only to clarify intention. However, for $\{\!|\bullet|\!\}$, and later $[\![\bullet]\!]$, the subscript *is* significant: $\{\!|\bullet|\!\}_r$ enables the confined memory mode, confining the expressions to the memory owned by r. Note that this rule is only applicable for any endpoint r, meaning singular endpoints e as well as an indexed family $F[i]$.

The rule CpIf adds a deadlock freedom assert, and then forwards the choreographic projection to the subexpression and sub-statements. The unanimous transformation function, which computes a verification condition for deadlock freedom, is further discussed in Section 4.5. As there is no endpoint context on the sub-statements, the confinement memory mode is not used here.

Rule CpComm encodes the sending of message H_{msg} to location H_{dst}, while transferring the channel invariant R_I from r to p. This is done as follows. First, the message value v is computed, confined to the memory of r. Then, the permissions in the channel invariant are removed using `exhale`, using the confined memory mode to ensure only permissions of r are removed. To determine the permissions to be exhaled, a substitution operation is applied. The notation $R_I(v,\ r,\ p)$ replaces, in R_I, every occurrence of `\msg` with v, `\sender` with r and `\receiver` with p. E.g. `\msg > 0` would become v `> 0` after substitution. This substituted invariant is added to the state of p using `inhale`, and then the value is written to the destination location.

4.3 Parameterized example

Figure 6 shows an example application of rule CpCommRange (Fig. 7). Note the use of the three-argument `Perm` predicate to indicate permissions with additional metadata (see Section 3.3). The `inhale`/`exhale` statements respectively repeat the `requires`/`ensures` expressions, and are hence abbreviated in this figure.

4.4 Parameterized projection rules

The transformation rules for parameterized choreographies are shown in Fig. 7. Rule CpCommRange encodes a parameterized communication between two

```
channel_invariant
  \msg == sum(ns, \sender.tid, \sender.n);
communicate
  nodes[i := 0..N-1].sum -> nodes[i+1].in;
```

(a) Input communication

```
assert (\forall int i, j = 0..N-1; i+1 == j+1 ==> i == j);
par (int i = 0..N-1)
 context Perm(ns[i].sum, ϵ, ns[i]) ** Perm(ns[i+1].in, 1, ns[i+1])
 requires ns[i].sum == sum(ns, ns[i].tid, ns[i].n);
 ensures ns[i+1].in == sum(ns, ns[i].tid, ns[i].n);
{ int v = ns[i].sum; exhale ...; inhale ...; ns[i+1].in = v; }
```

(b) Output PVL

Fig. 6: Encoding of a parameterized communicate from Fig. 3

endpoint families. It is essentially a regular communication, wrapped in a `par` block. This is crucial: if all message transfers can happen independently in parallel, they can be safely split up into separate threads, as is done by the endpoint projection. Before the `par` block, an `assert` is generated which checks injectivity of the expression d over the given range $[E_l, E_h)$ of the `communicate`. This assert ensures the sender-receiver relation is injective, i.e. for each sender there must be exactly one receiver, and vice versa.

Note how the syntax of message and destination fields is restricted: fields can only be dereferenced on an indexed family $F[i]$. This is in contrast with rule CpComm, where the object is a heap-dependent expression. This restriction ensures the entire process of projection stays automatic. Allowing a heap expression H would require user annotations. A workaround for the restriction is to assign a heap-dependent expression in a preceding method call.

In the generated par block, we require ϵ permission to read the message field f, which means the verifier will pick a positive fraction smaller than the available permission. Using ϵ ensures the location can only be read, and not written to, which allows the verifier to maintain that f does not change. The `context` keyword here is syntactic sugar for a symmetric `requires` and `ensures` clause. Rule CpCommRange performs substitution on R_I like rule CpComm. E.g. in the precondition of the `par` block, the notation $R_I(...)$ replaces, in R_I, `\msg` with $F[i].f$, `\sender` with $F[i]$ and `\receiver` with $G[d(i)]$.

Rule CpExprRange shows how to project an endpoint expression with a range. Essentially, `\endpoint` is replaced with a universal quantifier, and the inner expression E is confined to the memory accessible to $F[i]$.

Rule CpMethodCallRange encodes that a method call is executed in parallel on a range of an endpoint family. It does this by projecting the method call confined to a symbolic element of this endpoint family, $F[i]$, and wrapping that in a PVL `par` block. Wrapping the method call in a `par` block encodes that the methods must run in parallel and independently. The contract for the `par` block is taken from the method using $\mathsf{pre}(m, F[i])$ and $\mathsf{post}(m, F[i])$ to return the pre-/postcondition of m. They also replace any occurrence of `this` in the return value with the second argument, in this case $F[i]$. Note that only method calls directly on $F[i]$ are allowed, similar to rule CpCommRange. This is to keep projection automatic. A workaround for this restriction is to compute a heap-dependent expression within the method m, and call a method the result.

CpMethodCallRange

$$\{\!|\texttt{endpoint } F[i] \texttt{ := } E_l \texttt{ .. } E_h\texttt{]: } F[i].m();|\!\} =$$
$$\texttt{par (int } i \texttt{ = } E_l \texttt{ .. } E_h\texttt{)}$$
$$\texttt{requires } \{\!|\mathsf{pre}(m, F[i])|\!\}_{F[i]};$$
$$\texttt{ensures } \{\!|\mathsf{post}(m, F[i])|\!\}_{F[i]};$$
$$\texttt{\{ } \{\!|\texttt{endpoint } F[i]\texttt{: } F[i].m();|\!\} \texttt{ \}}$$

CpExprRange

$$\{\!|\texttt{(\textbackslash endpoint } F[i \texttt{ := } E_l \texttt{ .. } E_h]\texttt{; } E)|\!\} =$$
$$\texttt{(\textbackslash forall int } i \texttt{ = } E_l \texttt{ .. } E_h\texttt{; } \{\!|E|\!\}_{F[i]}\texttt{)}$$

CpCommRange

$$\left\{\!\!\left|\begin{array}{l}\texttt{channel\_invariant } R_I\texttt{(\textbackslash msg, \textbackslash sender, \textbackslash receiver);}\\ \texttt{communicate } F[i \texttt{ := } E_l \texttt{ .. } E_h]\texttt{: } F[i].f \texttt{ -> } G[d(i)]\texttt{: } G[d(i)].g\texttt{;}\end{array}\right|\!\!\right\} =$$
$$\texttt{assert (\textbackslash forall int i, j = } E_l \texttt{ .. } E_h\texttt{; } d(\texttt{i}) \texttt{ == } d(\texttt{j}) \texttt{ ==> i == j);}$$
$$\texttt{par (int } i \texttt{ = } E_l \texttt{ .. } E_h\texttt{)}$$
$$\texttt{context } \{\!|\texttt{Perm}(F[i].f, \epsilon)|\!\}_{F[i]} \texttt{ ** } \{\!|\texttt{Perm}(G[d(i)].g, 1)|\!\}_{G[d(i)]};$$
$$\texttt{requires } \{\!|R_I(F[i].f, F[i], G[d(i)])|\!\}_{F[i]};$$
$$\texttt{ensures } \{\!|R_I(G[d(i)].g, F[i], G[d(i)])|\!\}_{G[d(i)]};$$
$$\texttt{\{ } T \texttt{ } v \texttt{ = } \{\!|F[i].f|\!\}_{F[i]};$$
$$\texttt{exhale } \{\!|R_I(v, F[i], G[d(i)])|\!\}_{F[i]};$$
$$\texttt{inhale } \{\!|R_I(v, F[i], G[d(i)])|\!\}_{G[d(i)]};$$
$$\{\!|G[d(i)].g|\!\}_{G[d(i)]} \texttt{ = } v\texttt{; \}}$$

Fig. 7: Parameterized choreographic projection rules

4.5 Branch unanimity

Branch unanimity is defined through the function $\mathsf{unanimous}(E)$ and supporting functions, shown below. It evaluates to `true` if all endpoints in E evaluate E to the same result. We split this into two cases: either all α_i evaluate E to `true`, or all evaluate to `false`. Evaluating the condition for each individual endpoint occurring in E takes two steps. The first step is using the confined memory mode, which drops parts of the expression that are not relevant to the given endpoint. For example, $\{\!|\texttt{(\textbackslash endpoint a; } E_1\texttt{) \&\& (\textbackslash endpoint b; } E_2\texttt{)}|\!\}_{\mathsf{a}} = \{\!|E_1|\!\}_{\mathsf{a}}$.

The second step is wrapping E, confined to $F[i]$, in a universal quantifier. This enables reasoning over the entire endpoint family, even though during verification the size of endpoint families remains symbolic.

$$\mathsf{unanimous}(E) = \overbrace{(\mathsf{ground}(E, \alpha_1, \texttt{true}) \texttt{ \&\& } \cdots)}^{\text{for all } \alpha_1, \ldots, \alpha_n \in E} \texttt{ || } \overbrace{(\mathsf{ground}(E, \alpha_1, \texttt{false}) \texttt{ \&\& } \cdots)}^{\text{for all } \alpha_1, \ldots, \alpha_n \in E}$$
$$\mathsf{ground}(E, r, b) = \{\!|E|\!\}_r \texttt{ == } b$$
$$\mathsf{ground}(E, F[i \texttt{ := } E_l \texttt{ .. } E_h], b) = \texttt{(\textbackslash forall int } i \texttt{ = } E_l \texttt{ .. } E_h\texttt{; } \{\!|E|\!\}_{F[i]} \texttt{ == } b\texttt{)}$$

4.6 Choreographic projection of distributed summation

The choreographic projection of Fig. 3 is shown in Fig. 8. We focus on the `while` loop as it contains the core of the algorithm. Each statement is transformed by an application of the rules CpCommRange, CpMethodCallRange and CpExprRange. Note that the branch unanimity check is added as a loop invariant, instead of a separate assert, to ensure branch unanimity is checked both at loop entry and exit. The `false` branch has been ommitted using "...". As the arguments of the `exhale`/`inhale` statements on lines 8 and 9 just repeat the contract of the preceding par block, they have also been ommitted. For the parameterized method call on line 10 we inline the contract of `update`, resulting in only an `ensures` clause.

```
1  loop_invariant (\forall int i = 0..N; ns[i].n < N-1 == true) || ...;
2  loop_invariant (\forall int i = 0..N; ns[i].sum == sum(ns, ns[i].i, ns[i].n));
3  while ((\forall int i = 0..N; ns[i].n < N-1)) {
4    par (int i = 0..N-1)
5      context Perm(nodes[i-1].sum, ϵ) ** Perm(nodes[i].in, 1);
6      requires ns[i].sum == sum(ns, ns[i].i, ns[i].n);
7      ensures ns[i+1].in == sum(ns, ns[i].i, ns[i].n);
8    { int v = ns[i].sum; exhale ...; inhale ...; ns[i+1].in = v; }
9    int v = ns[N-1].sum; exhale ...; inhale ...; ns[0].in = v;
10   par (int i = 0..N)
11     ensures ns[i].sum = ns[i].in + ns[i].v;
12   { ns[i].update(); }
```

Fig. 8: Choreographic projection of core `while` loop of Fig. 3

5 Endpoint Projection

We will now discuss the transformation rules for the endpoint projection. It is written as $[\![\bullet]\!]_r$, where we refer to r as the *projection target*. The purpose of the endpoint projection is to transform a choreography such that it only executes parts relevant to the endpoint r. In this process, all choreographic primitives are replaced with plain PVL constructs. If this is done for all endpoints r in the choreography, when all endpoint projections are composed in parallel, the resulting program behaves exactly the same as the original choreography [11].

For non-parameterized choreographies, the endpoint projection can be done using a simple syntactic check [2]. Summarizing, for a given projection target r, simply retain all choreographic statements that mention r. Parameterized choreographies introduce endpoint families, whose sizes will only be known at runtime. This makes the endpoint projection challenging: how to determine if the endpoint $F[i]$ falls in the range `0..N`?

We resolve this by delaying projection of parameterized primitives until runtime. This is done by wrapping the projected statements in an `if` that checks if $F[i]$ is in the relevant range. If so, the statement is executed as if projected for $F[i]$; otherwise, it is skipped. This way, the endpoint projection can safely simulate the program for any possible endpoint $F[i]$, at the cost of including an extra `if`.

5.1 Non-parameterized example

To give an intuition for the endpoint projection, we show a concrete example. Figure 4a is the input choreography, and Fig. 9 shows the output PVL of the endpoint projection for both `a` and `b`. Each communication is transformed into a concrete channel operation, depending on whether the projection target is `a` or `b`. For the condition of `if` and endpoint statements, only those relevant to the current projection target are kept, while others are replaced by `true`.

5.2 Non-parameterized projection rules

In Fig. 10 (top) we summarize the conceptually interesting endpoint projection rules. The full listing can be found in [18].

```
1  chan1.writeValue(a.balance > a.n);
2  if (a.balance > a.n && true) {
3    chan2.writeValue(a.n);
4    a.balance = a.balance - a.n;
5    /* skip */
6  }
```

(a) Endpoint projection for `a`.

```
1  b.ok = chan1.readValue();
2  if (true && b.ok) {
3    b.n = chan2.readValue();
4    /* skip */
5    b.balance = b.balance + b.n;
6  }
```

(b) Endpoint projection for `b`.

Fig. 9: Endpoint projection of Fig. 4a.

Rule EpIf encodes an `if` statement by keeping it and applying the endpoint projection to the condition and sub-statements. For rule EpExpr, if the endpoint annotation matches the current projection target, the expression is simply kept.

Rule EpSend describes how a `communicate`, with implicit name L, should be encoded if singular endpoint a is in the sending position. Before the endpoint projection is done, VeyMont generates a channel instance for each `communicate` statement and assigns it to L. Then, when applying rule EpSend, the statement is replaced by $[\![L]\!]_a$`.writeValue(`E_{msg}`)`, where $[\![L]\!]_a$ represents the channel instance generated beforehand. The methods `readValue` and `writeValue` are part of the runtime environment that VeyMont generates automatically [2,19].

5.3 Parameterized projection rules

Parameterized projection rules are shown in Fig. 10 (bottom). These have to account for ranges of endpoint families. This is done by partially delaying the projection until runtime. For each projected statement or expression we check if the index of the current projection target is in the range (or the exact index) specified by the expression or statement. E.g. in rule EpRange, the expression E will only be evaluated if the index of the current projection target falls in the range of $[E_l, E_h)$. Similarly, rule EpRangeSend wraps the call to `writeValue` in an `if` statement to ensure adherence to the range. Rule EpRangeReceive is symmetric, in particular the use of d is also inverted as follows.

We require the function d used to compute the receiver index to be invertible. This is important, as the index of the sender determines which channel the receiving party should read from. We use the notation d^{-1} for the inverse function, e.g. in EpRangeReceive. The choreographic projection already guarantees that d is injective (Section 4.4). To actually compute the function d^{-1} in the projection, we use simple pattern matching to invert each operation in d. For example, if $d(i) = i + 1$, then $d^{-1} = i - 1$. This is an approach inspired by previous work: Ng et al. present Table II as a basis for such a transformation [16]. We think this step could be improved by reusing results in the field of bidirectional functions, such as [14]. We leave this for future work.

5.4 Example endpoint projection

The endpoint projection of Fig. 3 is shown in Fig. 11. There are two major differences between the choreography and its endpoint projection. First, each

EpIf
$[\![\texttt{if (}H\texttt{) } S_{true}\ S_{false}]\!]_r = \texttt{if (}[\![H]\!]_r\texttt{) } [\![S_{true}]\!]_r\ [\![S_{false}]\!]_r$

EpExpr
$[\![\texttt{(\textbackslash endpoint } e\texttt{; } E\texttt{)}]\!]_e = E$

EpSend
$[\![L\texttt{: communicate } a\texttt{: } H_{msg}\texttt{ -> } b\texttt{: } H_{dst}\texttt{;}]\!]_a^{send} =$
$[\![L]\!]_a\texttt{.writeValue(}H_{msg}\texttt{) with \{}$
$\texttt{sender = } a\texttt{; receiver = } b\texttt{; \};}$

EpRangeSend
$[\![L\texttt{: communicate } F\texttt{[}j\texttt{: } E_l\ \texttt{..}\ E_h\texttt{].}f\texttt{ -> } G\texttt{[}d(j)\texttt{].}g]\!]_{F[i]}^{send} =$
$\texttt{if (}E_l\texttt{ <= }i\texttt{ \&\& }i\texttt{ < }E_h\texttt{) \{}$
$[\![L]\!]_{F[i]}\texttt{[}i\texttt{].writeValue(}F\texttt{[}i\texttt{].}f\texttt{) with \{}$
$\texttt{sender = } F\texttt{[}i\texttt{]; receiver = } G\texttt{[}d(i)\texttt{]; \}; \}}$

EpRangeReceive
$[\![L\texttt{: communicate } F\texttt{[}j\texttt{: } E_l\ \texttt{..}\ E_h\texttt{].}f\texttt{ -> } G\texttt{[}d(j)\texttt{].}g]\!]_{G[i]}^{receive} =$
$\texttt{if (}E_l\texttt{ <= }d^{-1}(i)\texttt{ \&\& }d^{-1}(i)\texttt{ < }E_h\texttt{) \{}$
$G\texttt{[}i\texttt{].}g\texttt{ = }[\![L]\!]_{G[i]}\texttt{[}d^{-1}(i)\texttt{].readValue() with \{}$
$\texttt{sender = } F\texttt{[}d^{-1}(i)\texttt{]; receiver = } G\texttt{[}i\texttt{]; \}; \}}$

EpRange
$[\![\texttt{(\textbackslash endpoint } F\texttt{[}j\texttt{ := } E_l\ \texttt{..}\ E_h\texttt{]; } E\texttt{)}]\!]_{F[i]} = E_l\texttt{ <= }i\texttt{ \&\& }i\texttt{ < }E_h\texttt{ ==> } E$

Fig. 10: Non-parameterized and parameterized endpoint projection rules

communication is split up into write and read statements. This is because in a parameterized communication, an endpoint can be both a sender and a receiver. E.g. in the summation choreography, line 10 of Fig. 3, node i must send to node $i+1$, and receive from node $i-1$. This is in contrast to regular non-parameterized communication, in which case it is statically known if the projection target is *either* a sender or a receiver, meaning less code is generated.

The second difference is that each statement in Fig. 11 is wrapped in an `if`, ensuring that the action is only executed if the index of the current endpoint (in this case i) falls in the range specified by each `communicate` statement (e.g. line 3). When a `communicate` is not parameterized, but involves a parameterized endpoint, such an `if` is also necessary, e.g. on line 5. Finally, this transformation is applied similarly to parameterized method calls (line 7).

6 Related Work

VeyMont This paper builds on work around VeyMont. Jongmans et al. formalised verification of choreographies [11]. Van den Bos et al. first implemented the choreographic and endpoint projection in VeyMont [2]. Rubbens et al. extended them with shared memory support and annotation preservation [19].

Choreographies The following works do not consider choreographic verification, or shared memory, but they do concern parameterization. We expect that our insights can be applied to the following works, and vice versa.

Jongmans introduces *first-person choreographic programming* (1CP), which is a novel formulation of choreographies with parameterization [10]. It is event-driven and dynamic. They prove deadlock freedom of well-typed choreographies, support intricate messaging patterns such as pipelined communication, and provide tool support. They do not provide a way to verify functional correctness

```
1  loop_invariant 0 <= i && i < N ==> ns[i].sum == sum(ns, i, ns[i].n);
2  while (0 <= i && i < N ==> ns[i].n < N-1)) {
3    if (0 <= i && i < N-1) chan1[i].writeValue(ns[i].sum) with { sender = ns[i]; receiver = ns[i + 1]; };
4    if (0 + 1 <= i && i < N-1+1) ns[i].in = chan1[i - 1].readValue() with { sender = ns[i - 1]; receiver = ns[i]; };
5    if (i == N - 1) chan2.writeValue(ns[N-1].sum) with { sender = ns[N-1]; receiver = ns[0]; };
6    if (i == 0) ns[0].in = chan2.readValue() with { sender = ns[N-1]; receiver = ns[0]; };
7    if (0 <= i && i < N) ns[i].update(); }
```

Fig. 11: Example of endpoint projection of Fig. 3

of the choreographies, nor of the endpoint projections, and also do not support shared memory. A key difference between their and our formulation of parameterized choreographies is that they do not allow indexing into endpoint families. Instead, they fully describe the network topology before the choreography is started, avoiding the need to define indexing of endpoint families in their semantics. We avoid defining the network topology by using verification to ensure injectivity and bounds checking for indexing operations.

Bates et al. support parameterized choreographies in the tool MultiChor [1]. This is achieved through *census polymorphism*, which essentially parameterizes a choreography over a set of endpoints by leveraging the Haskell type system. They fully delay the endpoint projection until runtime, where we only do this for parameterized parts of a choreography. While conditions still need to be propagated between endpoints to maintain deadlock freedom, they employ *enclaves* to limit the scope in which conditions need to be propagated. We use branch unanimity to guarantee deadlock freedom at verification time. They also do not support endpoint family indexing directly, but instead fix the network topology during the initialization phase, similar to Jongmans [10].

Instead of generating programs from choreographies, Kjær et al. infer choreographies from parameterized programs [12]. They achieve this by parameterizing procedures with endpoint references, such that choreographies can model an endpoint substituting for another. Cruz-Filipe and Montesi similarly extend choreographies with procedures and dynamic participant allocation, allowing e.g. pipelined communication [5]. They do not support endpoint family indexing.

Session types Session types [21] are related, yet subtly different from choreographies. Session types type check protocol conformance for a given implementation, whereas choreographies allow generating an implementation. Related work in parameterization of session types focuses on the *multi-party* variant [9,20], which allows more than two parties in the session type.

None of the works on parameterized session types we found support verification of functional correctness and shared memory [7,4,3,16,8]. Their support for indexing into endpoint families is usually restricted to some decidable fragment of arithmetic, where we support general recursive functions. We use verification to show that bounds of endpoint families are respected, and that indexing is injective, guaranteeing that the endpoint projection produces safe code. Except for Hamers et al [8], they all require some form of symmetry in the session type to ensure deadlock-freedom of the projection, whereas we use branch unanim-

ity to guarantee deadlock freedom (Section 4.5). The works on session types do support more communication patterns, such as many-to-one or pipelined communication. This is still a challenge for our formulation of choreographies: more annotations will be required from the user, making this extension non-trivial.

Hamers et al. [8] do dynamic checking of session types. This means they do not need a mechanism like branch unanimity to avoid deadlocks. In exchange, a session type might crash because non-compliance is detected at run-time.

Ng et al. [16] introduce the endpoint family notation we use as well, which is in turn inspired by [13]. Furthermore, Ng et al. also require that indexing of endpoint families is an invertible operation. We generalize this requirement: for the choreographic projection, indexing merely needs to be injective. This suffices for verification, as VerCors can reason about injectivity. The endpoint projection requires an invertible expression, as indices need to be computable.

7 Conclusion

We proposed an extension of the automated verifier and code generator VeyMont for parameterized choreographies, opening the door for verification of choreographies with an arbitrary number of participants.

Adding parameterization support required improvements to several VeyMont components. We first defined the syntax of choreographies with parameterized primitives, such as endpoint families and parameterized communication. We then extended the choreographic projection with support for parameterized choreographies, which leverages the `par` block from VerCors. In addition, we restrict the input language of the choreographic projection such that verification annotations can be automatically generated. We also showed that deadlock freedom of parameterized choreographies can be checked by quantification over entire endpoint families using universal quantifiers.

Also, we extended the endpoint projection with support for endpoint families. The endpoint projection generates one program that works for each symbolic index of the family. This is implemented by checking the index at run-time, enabling only statements for the specified index.

We have illustrated and motivated our contribution by verifying a distributed summation choreography, included in the artefact [17]. To the best of our knowledge, there is no prior work on verification of parameterized choreographies.

Future work We will complete the implementation of the approach presented in this work, and further evaluate the approach with more case studies. Then, future work will go in several directions. We will investigate the possibility of "ghost" communication statements, which allow communicating ghost state and proof hints between endpoints with zero run-time overhead. Another direction is to make interactions more flexible, for example by allowing heap locations as indices for parameterized communication statements, and by adding pipelined and many-to-one communication. Finally, we wish to integrate the branch unanimity check into the endpoint projection. This would allow modification, reverification and further analysis of deadlock freedom of the generated code.

References

1. Bates, M., Kashiwa, S., Jafri, S., Shen, G., Kuper, L., Near, J.P.: Efficient, portable, census-polymorphic choreographic programming (2024). https://doi.org/10.48550/ARXIV.2412.02107, submitted at PLDI'25.
2. van den Bos, P., Jongmans, S.: Veymont: Parallelising verified programs instead of verifying parallel programs. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14000, pp. 321–339. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_19
3. Castro-Perez, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. Proc. ACM Program. Lang. **3**(POPL), 29:1–29:30 (2019). https://doi.org/10.1145/3290342, https://doi.org/10.1145/3290342
4. Charalambides, M., Dinges, P., Agha, G.: Parameterized concurrent multi-party session types. In: Kokash, N., Ravara, A. (eds.) Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012. EPTCS, vol. 91, pp. 16–30 (2012). https://doi.org/10.4204/EPTCS.91.2
5. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: Bouajjani, A., Silva, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017. Lecture Notes in Computer Science, vol. 10321, pp. 92–107. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_7
6. Darabi, S., Blom, S.C.C., Huisman, M.: A verification technique for deterministic parallel programs. In: Barrett, C.W., Davies, M.D., Kahsai, T. (eds.) NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10227, pp. 247–264 (2017). https://doi.org/10.1007/978-3-319-57288-8_17
7. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Log. Methods Comput. Sci. **8**(4) (2012). https://doi.org/10.2168/LMCS-8(4:6)2012
8. Hamers, R., Horlings, E., Jongmans, S.: The discourje project: run-time verification of communication protocols in clojure. Int. J. Softw. Tools Technol. Transf. **24**(5), 757–782 (2022). https://doi.org/10.1007/S10009-022-00674-Y
9. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
10. Jongmans, S.S.: First-person choreographic programming with continuation-passing communications. In: 34th European Symposium on Programming, ESOP 2025 (2025), accepted for ESOP 2025
11. Jongmans, S., van den Bos, P.: A predicate transformer for choreographies - computing preconditions in choreographic programming. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings.

Lecture Notes in Computer Science, vol. 13240, pp. 520–547. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_19
12. Kjær, B.A., Cruz-Filipe, L., Montesi, F.: From infinity to choreographies - extraction for unbounded systems. In: Villanueva, A. (ed.) Logic-Based Program Synthesis and Transformation - 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13474, pp. 103–120. Springer (2022). https://doi.org/10.1007/978-3-031-16767-6_6
13. Magee, J., Kramer, J.: Concurrency - state models and Java programs (2. ed.). Wiley (2006)
14. Matsuda, K., Wang, M.: "Bidirectionalization for free" for monomorphic transformations. Sci. Comput. Program. **111**, 79–109 (2015). https://doi.org/10.1016/J.SCICO.2014.07.008
15. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023). https://doi.org/10.1017/9781108981491
16. Ng, N., Yoshida, N.: Pabble: parameterised scribble. Serv. Oriented Comput. Appl. **9**(3-4), 269–284 (2015). https://doi.org/10.1007/S11761-014-0172-8
17. Rubbens, R., Van den Bos, P., Huisman, M.: Artefact of: Verified parameterized choreographies. https://doi.org/10.5281/zenodo.14900264
18. Rubbens, R., van den Bos, P., Huisman, M.: Verified parameterized choreographies technical report (2025). https://doi.org/10.48550/arXiv.2502.15382
19. Rubbens, R., van den Bos, P., Huisman, M.: Veymont: Choreography-based generation of correct concurrent programs with shared memory. In: Kosmatov, N., Kovács, L. (eds.) Integrated Formal Methods. pp. 217–236. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-76554-4_12
20. Yoshida, N., Gheri, L.: A very gentle introduction to multiparty session types. In: Hung, D.V., D'Souza, M. (eds.) Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings. Lecture Notes in Computer Science, vol. 11969, pp. 73–93. Springer (2020). https://doi.org/10.1007/978-3-030-36987-3_5
21. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In: Fernández, M., Kirchner, C. (eds.) Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006. Electronic Notes in Theoretical Computer Science, vol. 171, pp. 73–93. Elsevier (2006). https://doi.org/10.1016/J.ENTCS.2007.02.056