

# Verified Parameterized Choreographies

## Technical Report

Robert Rubbens<sup>(✉)</sup>  Petra van den Bos   
 Marieke Huisman 

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

### Abstract

This technical report contains the full set of definitions and projection rules of the paper “Verified Parameterized Choreographies” by Rubbens et al. [4]. It also supplements the artefact [3].

## 1 Complete formal syntax

The figure below is the formal syntax of PVL programs supported by VeyMont, including the complete OOP fragment of PVL.

**Core PVL** The syntax of PVL is shown in Fig. 1, of which the left is the syntax for core OOP PVL. PVL has classes, methods, fields, and supports several built-in types, such as integers (`int`), booleans, and sequences (e.g. `seq<int>`). It supports standard statements such as `while`, `if` and variable assignment, and standard expressions such as boolean logic and arithmetic. It also supports verification primitives such as contracts with pre- and postconditions, and assertions and ownership through permission annotations. The main primitive for concurrency in PVL is the `par` block. When a main thread reaches a `par` block,  $N$  subthreads are spawned to execute the body of the `par` block in parallel. The main thread waits until all subthreads are finished, and then continues with the remainder of the program.

We want to highlight how expressions are defined in Fig. 1. Pure expressions  $E$  only depend on local variables and immutable value constructors, such as sequences and sets. Heap-dependent expressions  $H$  are a superset of  $E$ , that can also refer to fields of objects. Resource expressions ( $R$ ) are a superset of  $H$ , and include permissions using the `Perm` keyword, as well as the separating conjunction operator `**` to compose resources. These different kinds of expressions give rise to variations of several other nodes: pure contracts ( $K$ ), contracts that only inspect the heap and not modify it ( $K_H$ ) and contracts that require and return resources ( $K_R$ ), pure functions ( $f$ ) and functions that read the heap ( $f_H$ ).

**Choreography DSL in PVL** On the right of Fig. 1 is the choreographic fragment of PVL. A choreography has zero or more parameters, and defines one or more endpoint or endpoint families. Endpoint  $e$  are singular endpoints, defined to have a name and instructions on how it should be instantiated with a given constructor. Endpoint families extend this notion with an extra size parameter that indicates the size of the endpoint family at runtime. Where a singular endpoint is represented at run-time with an instance of the given class  $C$ , an endpoint family is represented with an immutable sequence of such instances. Finally, a choreography also contains a `run` declaration, which is the main body of the choreography and contains a sequence of choreographic statements.

$x, y, z ::= \text{field}, \quad v, u, w ::= \text{variable}, \quad m ::= \text{method},$   
 $f ::= \text{function}, \quad C ::= \text{class}, \quad P ::= \text{predicate},$   
 $T ::= \text{int} \mid \text{boolean} \mid \text{seq}\langle T \rangle \mid C \mid \dots$   
 $K ::= \text{requires } E; \text{ ensures } E;$   
 $K_H ::= K \text{ with } R, H \quad K_R ::= K \text{ with } R, R$   
**prog**  $::= \overline{\text{decl}}$   
**decl**  $::= \text{class } C \{ \overline{D_{cls}} \} \mid \text{resource } P(\overline{Tv}) = E;$   
 $\mid K \text{ pure } T \ f(\overline{Tv}) = E; \mid K_H \text{ pure } T \ f_H(\overline{Tv}) = H;$   
 $\mid \text{chor}$   
 $D_{cls} ::= T \ x; \mid K_R \ T \ m(\overline{Tv}) \ S_m$   
 $E ::= v \mid r \mid F \mid E + E \mid E \ \&\& \ E \mid E.f(\overline{E}) \mid \dots$   
 $H ::= E \text{ extended with: } E_h.x \mid E_h.f_h(\overline{E_h}) \mid F[E_h] \mid \text{this} \quad S_{ep} ::= H.m(\overline{H}) \mid H := H;$   
 $R ::= H \mid \text{Perm}(H.x, H) \mid R \ ** \ R \mid P(\overline{H})$   
 $\mid H ==> R \mid \text{QP}$   
 $S_m ::= \text{assert } H; \mid H = H; \mid H.m(\overline{H})$   
 $\mid \text{inhale } R; \mid \text{exhale } R; \mid \text{if } (H) \ S_m \ S_m$   
 $\mid \text{loop\_invariant } R; \text{ while } (H) \ S_m \mid \dots$   
 $\mid K_R \text{ par } (T \ v = H \ .. \ H) \ S_m$   
 $e, a, b ::= \text{endpoint} \quad F, G ::= \text{endpoint family},$   
**chor**  $::= K_R \text{ choreography}(\overline{Tv}) \{ \overline{D_{chor}} \}$   
 $D_{chor} ::= \text{endpoint } e = C(\overline{H});$   
 $\mid \text{endpoint } F[v := 0 \ .. \ H] = C(\overline{E});$   
 $\mid K_R \text{ run } \{ S_{chor} \}$   
 $S_{chor} ::= \text{if } (H_{chor}) \ S_{chor} \ S_{chor} \mid \text{assert } R_{chor};$   
 $\mid \text{loop\_invariant } R_{chor}; \text{ while } (H_{chor}) \ S_{chor}$   
 $\mid \text{endpoint } \alpha: S_{ep}$   
 $\mid \text{channel\_invariant } R_{chan}; \text{ communicate } \alpha: H \rightarrow \alpha: H;$   
 $H_{chor} ::= (\backslash \text{endpoint } \alpha; H) \mid H_{chor} \ \&\& \ H_{chor}$   
 $R_{chor} ::= (\backslash \text{endpoint } \alpha; R) \mid R_{chor} \ \&\& \ R_{chor} \mid (\backslash \text{chor } H)$   
 $R_{chan} ::= R \text{ extended with: } \backslash \text{msg} \mid \backslash \text{sender} \mid \backslash \text{receiver}$   
 $r, p ::= e \mid F[E] \quad \alpha, \beta ::= r \mid F[v := E \ .. \ E]$

Figure 1: PVL syntax. Left: OOP fragment, right: choreographic fragment.

There are two ways to refer to endpoints. First, to refer to singular endpoints there is the notation  $r$ , which refers to either an endpoint  $e$  or a member of an endpoint family  $F$  at index  $E$ . Second, there is the notation  $\alpha$  for endpoint targets in general, which extends singular endpoints with ranges of endpoint families. This can be used to state that e.g. a statement must be executed by a subrange of an endpoint family.

The syntax for choreographic statements  $S_{chor}$  partially overlaps with regular PVL statements  $S_m$ , but they cannot be used interchangeably. Specifically because e.g. the choreographic **if** requires its condition to be one or more endpoint expressions  $(\backslash \text{endpoint } \alpha; H)$  combined with  $\&\&$ , whereas the regular PVL **if** requires a plain heap expression  $H$ . The semantics is similar to the semantics of each statement in PVL, except that each endpoint only executes those statements that are relevant to the endpoint. For example, the choreographic statement **endpoint**  $a: e.x := 3$  will be executed by  $a$ , but skipped by  $b$ . Composite statements are transparent with regards to this; if an endpoint does not occur within a composite choreographic statement, it skips it.

A communication statement is parameterized when the  $\alpha$  notation is used to denote a range, such as  $F[v := 0 \ .. \ N]$ . When both alphas of the **communicate** indicate a singular endpoint, it is just a regular non-parameterized **communicate** (though even in the non-parameterized **communicate**, a member of an endpoint family might participate through endpoint family indexing, such as  $F[5]$ ).

The user can also declare a channel invariant on a **communicate** statement, which specifies an invariant over values sent over that channel. This invariant must be proven over the values sent, and may be assumed over the values received. Within this invariant, the keywords **\sender**, **\receiver** and **\msg** must be used to refer symbolically to the respective concepts.

Choreographic expressions are adapted to allow the same projectability that

$$\begin{array}{lll}
\text{SORT} & & \\
& \text{sort}(e) & = e \\
& \text{sort}(F[i]) & = F \\
& \text{sort}(F[i := E_{low} \dots E_{high}]) & = F
\end{array}$$

Figure 2: Definition of `sort` to approximate inequality between instances of  $\alpha$

is natural for statements. The primitive here is the endpoint expression, written as  $(\backslash\text{endpoint } \alpha; H)$ , and also for  $R$ . This expression indicates that  $H$  is only relevant for the endpoint target  $\alpha$ , and must be ignored by other endpoints. This means that any endpoint covered by  $\alpha$  will evaluate the expression, while endpoints not covered by  $\alpha$  will simply continue as if the expression evaluated to `true`. This is sound, because endpoint expressions may only occur positively, and because VeyMont checks branch unanimity [1].

The `\chor` expression is used to indicate that an expression should only be included in the choreographic projection, and that it should not be included in the endpoint projection. In addition, within a `\chor` expression, permissions from all endpoints can freely be mixed. In spirit, `\chor` is similar to `assume`, in that it is used to “debug” non-verifying programs, and that any use of `\chor` should include an explanation of why it is needed, or otherwise removed. For more information on endpoint expressions, we refer the reader to [2].

## 2 Auxiliary definitions

Figure 2 shows the auxiliary definition `sort`. The function `sort` approximates inequalities on  $\alpha$ . This is useful for checking if it is possible for two  $\alpha$  notations to be equal. For example, if `sort(F[i])`  $\neq$  `sort(G[j])`, then  $F[i]$  and  $G[j]$  are also distinct.

We define functions `pre(m, E)` and `post(m, E)` axiomatically to return the pre-/postcondition of  $m$ . In addition, they replace any occurrence of `this` in the return value with  $E$ . If these functions are given a class  $C$ , they return the pre-/postcondition of the constructor of the class.

## 3 Choreographic Projection Rules

We will now discuss the transformation rules for the choreographic projection.

The transformation rules distinguish between 1. singular endpoints using  $e$ , 2. singular or indexed endpoint families using  $r$ , and 3. endpoints or endpoint family ranges using  $\alpha$ . E.g. rule `CPEXPR` applies only to singular endpoints, such that another rule (in this case rule `CPEXPRRANGE`) is necessary to handle the parameterized case. In contrast, e.g. rule `CPEXPRSKIP` uses  $\alpha$ , and hence works for both endpoint family ranges as well as singular endpoints.

Rule `CPEXPR` enables confined memory mode [2] to make sure  $E$  is evaluated using only memory of  $r$ .

Rule `CPEXPRSKIP` skips an expression by transforming it to `true` if it is not relevant for the current target for confinement. This is safe, because the side condition `sort( $\alpha$ )`  $\neq$  `sort( $r$ )` guarantees that the expression is not relevant to  $r$ .

Rule `CPASSIGN` uses confined memory mode to ensure the assignment is executed on the memory of  $r$ . For this rule there is no parameterized version. This is because it is difficult to automatically infer the footprint of the expressions  $E_{loc}$  and  $E_v$  in a parameterized context. If required, the user can work around this by defining a method on an endpoint that only writes to a field, and call this using the rule for parameterized method invocation, discussed later.

$$\begin{array}{lll}
\text{CPEXPR} & \llbracket (\backslash \text{endpoint } r; E) \rrbracket = \llbracket E \rrbracket_r & \text{CPEXPRSKIP} \quad \llbracket (\backslash \text{endpoint } \alpha; E) \rrbracket_r = \text{true} \\
& & \text{if } \text{sort}(\alpha) \neq \text{sort}(r) \\
& & \text{CPASSIGN} \quad \llbracket r: H_{loc} := H_v; \rrbracket = \llbracket H_{loc} \rrbracket_r = \llbracket H_v \rrbracket_r; \\
\text{CPIF} & \llbracket \text{if } (H) S_{\text{true}} S_{\text{false}} \rrbracket = \{ \text{assert unanimous}(H); \\
& \quad \text{if } (\llbracket H \rrbracket) \llbracket S_{\text{true}} \rrbracket \llbracket S_{\text{false}} \rrbracket \} & \text{CPWHILE} \quad \llbracket \text{loop\_invariant } R_{inv}; \text{while } (H_{cond}) S \rrbracket = \\
& & \quad \text{loop\_invariant unanimous}(R_{inv}) \ \&\& \ \llbracket R_{inv} \rrbracket; \\
& & \quad \text{while } (\llbracket H_{cond} \rrbracket) \llbracket S \rrbracket \\
\text{CPMETHODCALL} & \llbracket \text{endpoint } r: H.m(); \rrbracket = \llbracket H \rrbracket_r. \llbracket m() \rrbracket_r; & \text{CPCOMM} \quad \left\{ \begin{array}{l} \text{channel\_invariant } R_I(\backslash \text{msg}, \backslash \text{sender}, \backslash \text{receiver}); \\ \text{communicate } r: H_{msg} \rightarrow p: H_{dst}; \\ \{ T \ v = \llbracket H_{msg} \rrbracket_r; \\ \quad \text{exhale } \llbracket R_I(v, r, p) \rrbracket_r; \\ \quad \text{inhale } \llbracket R_I(v, r, p) \rrbracket_p; \\ \quad \llbracket H_{dst} \rrbracket_p = v; \} \end{array} \right\} = \\
\text{CPEXPRRANGE} & \llbracket (\backslash \text{endpoint } F[i := E_l \dots E_h]; E) \rrbracket = \\
& \quad (\backslash \text{forall int } i = E_l \dots E_h; \llbracket E \rrbracket_{F[i]}) & \text{CPEXPRINDEX} \quad \llbracket (\backslash \text{endpoint } F[j := E_l \dots E_h]; E(j)) \rrbracket_{F[i]} = \\
& & \quad E_l \leq i \ \&\& \ i < E_h \implies \llbracket E(i) \rrbracket_{F[i]} \\
& & \text{CPMETHODCALLRANGE} \quad \llbracket \text{endpoint } F[i := E_l \dots E_h]: F[i].m(); \rrbracket = \\
& & \quad \text{par (int } i = E_l \dots E_h) \\
& & \quad \quad \text{requires } \llbracket \text{pre}(m, F[i]) \rrbracket_{F[i]}; \\
& & \quad \quad \text{ensures } \llbracket \text{post}(m, F[i]) \rrbracket_{F[i]}; \\
& & \quad \{ \llbracket \text{endpoint } F[i]: F[i].m(); \rrbracket \} \\
& & \text{CPCOMMRANGE} \quad \left\{ \begin{array}{l} \text{channel\_invariant } R_I(\backslash \text{msg}, \backslash \text{sender}, \backslash \text{receiver}); \\ \text{communicate } F[i := E_l \dots E_h]: F[i].f \rightarrow G[d(i)]: G[d(i)].g; \end{array} \right\} = \\
& & \quad \text{assert } (\backslash \text{forall int } i, j = E_l \dots E_h; d(i) == d(j) \implies i == j); \\
& & \quad \text{par (int } i = E_l \dots E_h) \\
& & \quad \quad \text{context } \llbracket \text{Perm}(F[i].f, \epsilon) \rrbracket_{F[i]} \ \&\& \ \llbracket \text{Perm}(G[d(i)].g, 1) \rrbracket_{G[d(i)]}; \\
& & \quad \quad \text{requires } \llbracket R_I(F[i].f, F[i], G[d(i)]) \rrbracket_{F[i]}; \\
& & \quad \quad \text{ensures } \llbracket R_I(G[d(i)].g, F[i], G[d(i)]) \rrbracket_{G[d(i)]}; \\
& & \quad \{ T \ v = \llbracket F[i].f \rrbracket_{F[i]}; \\
& & \quad \quad \text{exhale } \llbracket R_I(v, F[i], G[d(i)]) \rrbracket_{F[i]}; \\
& & \quad \quad \text{inhale } \llbracket R_I(v, F[i], G[d(i)]) \rrbracket_{G[d(i)]}; \\
& & \quad \quad \llbracket G[d(i)].g \rrbracket_{G[d(i)]} = v; \}
\end{array}$$

Figure 3: All choreographic projection rules

The rules **CPIF** and **CPWHILE** forward the choreographic projection to their subparts, while also adding deadlock freedom checks [1]. As there is no endpoint context on these statements, no confinement is necessary.

Rule **CPMETHODCALL** evaluates the target of the method in confined memory mode. On then target, it calls a version of the method  $m$  adapted to the stratified permissions memory model [2].

Rule **CPCOMM** encodes a communication from endpoint  $r$  to endpoint  $p$ . First, the message value is computed, confined to the memory of  $r$ . Then, the channel invariant is removed from the state of  $r$  using the **exhale** statement. Note that the channel may contain the placeholder expressions  $\backslash \text{msg}$ ,  $\backslash \text{sender}$ ,  $\backslash \text{receiver}$ , in this case referring to the value of  $H_{msg}$ ,  $r$  and  $p$  respectively. The projection instantiates these placeholders with their concrete values by passing

$v$ ,  $r$  and  $p$  as arguments to the channel invariant  $R_I$ . Then, the invariant is added to the state of  $p$ , after which finally the value is written to the destination location.

Rule **CPEXPRRANGE** evaluates an expression for all endpoints in an endpoint family symbolically by replacing the `\endpoint` keyword with `\forall`. This is sound, as `\endpoint` expressions can only occur in a positive positions:  $H_{chor}$  is essentially a list of `\endpoint` expressions combined with `&&`. This is at the logical level equivalent to using `\forall`.

Rule **CPEXPRINDEX** shows how to project an endpoint expression with a range in confined mode: an implication is prepended to the expression  $E$  that ensures  $E$  is only evaluated if the confinement target index  $j$  is within the bounds of the endpoint expression range,  $E_{low}$  and  $E_{high}$ . Effectively, we intersect the range specified by the endpoint expression with the confinement target. This rule is necessary when using the confined memory mode for branch unanimity (i.e. the `unanimous` function).

Rule **CPMETHODCALLRANGE** transforms a method call on a range of endpoints into a `par` block that executes the method calls independently and in parallel. This is essential: if the `par` block can be proven correct, this means the method calls can safely be executed independently and in parallel, which means splitting this method call up using the endpoint projection is safe. The syntax for the object on which the method is called is restricted: instead of a general expression  $H$  we allow only an indexed family. This ensures the required annotations for the `par` block can be automatically generated, as it keeps the footprint predictable and exact.

Rule **CPCOMMRANGE** does something similar as rule **CPMETHODCALLRANGE**, except for two things. First, the injectivity is checked by adding an `assert` and a quantifier encoding the injectivity property over the expression  $d$ . Second, by modelling the actual message exchange within the `par` block. This message exchange works as follows: 1. evaluate the message in the context of  $F[i]$ , 2. remove the channel invariant from the state of  $F[i]$ , 3. add the state to  $G[d(i)]$ , 4. assign the message to the destination location, allowing only memory to be used of the receiving party. Similar to rule **CPMETHODCALLRANGE**, if this `par` block can be verified, it is safe to split this block across endpoints using the endpoint projection. For this rule, the allowed syntaxes for the message and destination are similarly restricted as **CPMETHODCALLRANGE** to allow for automatic annotation generation.

## 4 Endpoint Projection Rules

Rules **EPASSIGN**, **EPEXPR**, **EPAND**, **EPIF**, **EPWHILE** and their `*SKIP` versions should be self explanatory: they preserve the meaning of the choreographic statement if it is related to the current projection target, and otherwise replace it with the empty block statement (resp. `true` for expressions).

Rule **EPCOMM** shows that each communication statement is processed twice: once in *send* mode and once in *receive* mode, passed as an argument through the superscript position. The sending part is processed first to ensure the projected program cannot get stuck. Rule **EPCOMMSKIP** shows what happens when the projection target is neither in the sending or the receiving position: it is replaced with the empty block statement. The `sort` function is used here to avoid having to add duplicate cases for both singular endpoints  $e$  as well as endpoint family indices such as  $F[i]$ .

Rules **EPSEND** and **EPRECEIVE** shows that sends and receives are encoded with resp. `writeValue` and `readValue` method calls on a channel. The specific channel is retrieved from a table that is pre-generated similar to how this is done for the choreographic projection. This is indicated with the  $\llbracket L \rrbracket_r$  notation.

Rule **EPEXPRINDEX** shows how to transform an endpoint expression when

<b>EPASSIGN</b> $\llbracket e: H_{loc} := H_v; \rrbracket_e = H_{loc} = H_v;$	<b>EPASSIGNSKIP</b> $\llbracket e: H_{loc} := H_v; \rrbracket_r = \{\}$	<b>EPSEND</b> $\llbracket L: \text{communicate } a: H_{msg} \rightarrow b: H_{dst}; \rrbracket_a^{send} = \llbracket L \rrbracket_a.\text{writeValue}(H_{msg}) \text{ with } \{ \text{sender} = a; \text{receiver} = b; \};$
<b>EPRECEIVE</b> $\llbracket L: \text{communicate } a: H_{msg} \rightarrow b: H_{dst}; \rrbracket_b^{receive} = H_{dst} = \llbracket L \rrbracket_b.\text{readValue}() \text{ with } \{ \text{sender} = a; \text{receiver} = b; \};$	<b>EPCOMM</b> $\llbracket L: \text{channel\_invariant } R_{inv}; \text{communicate } \alpha: H_{msg} \rightarrow \beta: H_{dst}; \rrbracket_r = \llbracket L: \text{communicate } \alpha: H_{msg} \rightarrow \beta: H_{dst}; \rrbracket_r^{send}; \llbracket L: \text{communicate } \alpha: H_{msg} \rightarrow \beta: H_{dst}; \rrbracket_r^{receive};$	
<b>EPCOMMSKIP</b> $\llbracket L: \text{communicate } \alpha: H_{msg} \rightarrow \beta: H_{dst}; \rrbracket_r = \{\}$ if $\text{sort}(\alpha), \text{sort}(\beta), \text{sort}(r)$ pairwise distinct	<b>EPEXPR</b> $\llbracket (\backslash \text{endpoint } e; E) \rrbracket_e = E$	<b>EPEXPRSKIP</b> $\llbracket (\backslash \text{endpoint } \alpha; E) \rrbracket_r = \text{true}$ if $\text{sort}(\alpha) \neq \text{sort}(r)$
<b>EPEXPRINDEX</b> $\llbracket (\backslash \text{endpoint } F[E_i]; E) \rrbracket_{F[i]} = i == E_i ==> E$	<b>EPRANGE</b> $\llbracket (\backslash \text{endpoint } F[j := E_l \dots E_h]; E) \rrbracket_{F[i]} = E_l \leq i \ \&\& \ i < E_h ==> E$	
<b>EPAND</b> $\llbracket E_1 \ \&\& \ E_2 \rrbracket_r = \llbracket E_1 \rrbracket_r \ \&\& \ \llbracket E_2 \rrbracket_r$	<b>EPCHOR</b> $\llbracket (\backslash \text{chor } E) \rrbracket_r = \text{true}$	<b>EPIF</b> $\llbracket \text{if } (H) \ S_{true} \ S_{false} \rrbracket_r = \text{if } (\llbracket H \rrbracket_r) \ \llbracket S_{true} \rrbracket_r \ \llbracket S_{false} \rrbracket_r$
<b>EPWHILE</b> $\llbracket \text{loop\_invariant } R_i; \text{while } (H) \ S \rrbracket_r = \text{loop\_invariant } \llbracket R_i \rrbracket_r; \text{while } (\llbracket H \rrbracket_r) \ \llbracket S \rrbracket_r$	<b>EPINDEXSEND</b> $\llbracket L: \text{communicate } F[E_j]: H_{msg} \rightarrow r: H_{dst}; \rrbracket_{F[i]}^{send} = \text{if } (i == E_j) \{ \llbracket L \rrbracket_{F[i]}. \text{writeValue}(\llbracket H_{msg} \rrbracket_{F[i]}) \text{ with } \{ \text{sender} = F[i]; \text{receiver} = r; \}; \}$	
<b>EPINDEXRECEIVE</b> $\llbracket L: \text{communicate } r: H_{msg} \rightarrow F[E_j]: H_{dst}; \rrbracket_{F[i]}^{receive} = \text{if } (i == E_j) \{ \llbracket H_{dst} \rrbracket_{F[i]} = \llbracket L \rrbracket_{F[i]}. \text{readValue}() \text{ with } \{ \text{sender} = r; \text{receiver} = F[i]; \}; \}$	<b>EPRANGESEND</b> $\llbracket L: \text{communicate } F[j: E_l \dots E_h].f \rightarrow G[d(j)].g \rrbracket_{F[i]}^{send} = \text{if } (E_l \leq i \ \&\& \ i < E_h) \{ \llbracket L \rrbracket_{F[i]}[i]. \text{writeValue}(F[i].f) \text{ with } \{ \text{sender} = F[i]; \text{receiver} = G[d(i)]; \}; \}$	
<b>EPRANGERECEIVE</b> $\llbracket L: \text{communicate } F[j: E_l \dots E_h].f \rightarrow G[d(j)].g \rrbracket_{G[i]}^{receive} = \text{if } (E_l \leq d^{-1}(i) \ \&\& \ d^{-1}(i) < E_h) \{ G[i].g = \llbracket L \rrbracket_{G[i]}[d^{-1}(i)]. \text{readValue}() \text{ with } \{ \text{sender} = F[d^{-1}(i)]; \text{receiver} = G[i]; \}; \}$		

Figure 4: All endpoint projection rules

it concerns an indexed endpoint family, and when the current projection target is also an indexed endpoint family. In this case, the expression  $E$  is encoded in such a way that it is only evaluated if the indices match of the two endpoint families.

For rule EPRANGE, this is similar, except that the current projection target index now has to be in a range  $[E_h, )$ .

Rule EPCHOR always drops the expression  $E$ , as this annotation indicates the expression should only be included in the choreographic projection.

Rules EPINDEXSEND, EPINDEXRECEIVE, EPRANGESEND and EPRANGERECEIVE apply the same trick as EPEXPRINDEX, they check if the current projection target index falls in the range specified by the statement.

For rule EPRANGERECEIVE there is an additional complication: the inverse of the expression  $d$  needs to be computed at run-time to determine the sending endpoint index. While injectivity of this expression is checked using the chore-

ographic projection, this does not result in the actual expression  $d^{-1}$ ; VeyMont reasons symbolically about it during the choreographic projection, for the purposes of verification. Instead, the endpoint projection uses pattern matching to ensure  $d$  is in a form that is actually invertible. For example, the expression  $i + 1$  can be pattern matched to find that the inverted form is  $i - 1$ .

## References

- [1] Petra van den Bos and Sung-Shik Jongmans. “VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs”. In: *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Vol. 14000. Lecture Notes in Computer Science. Springer, 2023, pp. 321–339. DOI: [10.1007/978-3-031-27481-7\\_19](https://doi.org/10.1007/978-3-031-27481-7_19).
- [2] Robert Rubbens, Petra van den Bos, and Marieke Huisman. “VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory”. In: *Integrated Formal Methods*. Ed. by Nikolai Kosmatov and Laura Kovács. Cham: Springer Nature Switzerland, 2025, pp. 217–236. DOI: [10.1007/978-3-031-76554-4\\_12](https://doi.org/10.1007/978-3-031-76554-4_12).
- [3] Robert Rubbens, Petra Van den Bos, and Marieke Huisman. *Artefact of: Verified Parameterized Choreographies*. DOI: [10.5281/zenodo.14900264](https://doi.org/10.5281/zenodo.14900264).
- [4] Robert Rubbens, Petra Van den bos, and Marieke Huisman. *Verified Parameterized Choreographies*. Accepted at COORDINATION 2025.