VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory[⋆]

Robert Rubbens(✉), Petra van den Bos, and Marieke Huisman

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

Abstract. In the VeyMont tool, choreographies can be used to specify concurrent programs using a sequential format. To support choreography-based development, VeyMont verifies a given choreography for functional correctness and memory safety, and subsequently generates a correct concurrent program. However, the initial version of VeyMont did not support programs with shared memory. This paper shows how we overcome this limitation, by adding support for ownership annotations to VeyMont. Moreover, we also adapted the concurrent program generation, so that it does not only generate code, but also annotations. As a result, further changes and optimizations of the concurrent program can directly be verified. We demonstrate the extended capabilities of VeyMont on illustrative case studies.

Keywords: Deductive verification · Choreographies · Concurrent programs

1 Introduction

In program verification, auto-active verifiers prove correctness of programs automatically, with respect to a given specification. Writing specifications is non-trivial already in a sequential setting, and concurrency makes it even more challenging, as a concurrent program has a combinatorial number of interleavings to be considered. VeyMont [7] addressed this problem by combining choreographies [18], for specifying (concurrent) protocols, with deductive verification. VeyMont generates implementations for choreographies, which can be verified by using the VerCors verifier for concurrent software [3] as back-end. The choreographies of VeyMont allow specifying a concurrent program in a sequential format to ease verification, and then to generate the correct concurrent program.

In its purest form, a choreography [18] describes a sequence of message exchanges between participants of the choreography, called endpoints. The ordering of messages is partially fixed: an endpoint skips exchanges it does not participate in. Choreographies are deadlock free on the message level, meaning no endpoint will be stuck waiting for a message that will never be sent. Note

⋆ Supported by the NWO VICI 639.023.710 Mercedes project.

```
 1  choreography incrField() {
 2    endpoint a = Role();
 3    endpoint b = Role();
 4
 5    requires a.x > 0;
 6    ensures a.x == b.x;
 7    ensures a.x > 2;
 8    run {
 9      a.x := a.x + 1;
10      communicate a.x -> b.x;
11      b.x := b.x + 1;
12      communicate b.x -> a.x;
13    }
14  }
```

(a) A shared field is simulated by broadcasting intermediate results between `a` and `b`.

```
 1  choreography incrStore(Store s) {
 2    endpoint a = Role(s);
 3    endpoint b = Role(null);
 4
 5    requires a.s.x > 0;
 6    ensures a.s.x > 2;
 7    run {
 8      a.s.x := a.s.x + 1;
 9      // Store reference sent to b
10      communicate a.s -> b.s;
11      b.s.x := b.s.x + 1;
12      // Store reference sent to a
13      communicate b.s -> a.s;
14    }
15  }
```

(b) A shared field is incremented by both `a` and `b`. The messages function as barriers.

Fig. 1: Two choreographies where endpoints `a` and `b` each increment a value.

that in choreographies with shared memory and local actions, as presented in this work, deadlock in general is still possible. This is because endpoints can take local actions, such as acquiring locks, which might deadlock. Choreographies also guarantee that messages are well typed, meaning an endpoint will never receive an `int` when they are expecting a `float`. Finally, for each endpoint a specialized implementation can be generated. When these implementations are executed in parallel, messages are exchanged between processes as specified in the choreography. While similar, choreographies differ from session types [13]: a session type can only be used to type check implementations that are written by a user, choreographies allow automatic derivation of an implementation for each of its endpoints.

In VeyMont [7], a choreography specifies a concurrent program, such that its implementation can be generated. VeyMont supports verification of memory safety and functional correctness of these choreographies, which allows reasoning about e.g. program state properties. Such reasoning is not supported for the messages of traditional choreographies, which do not have local actions and shared memory. To support this verification, VeyMont requires users to annotate choreographies with contracts for functional correctness, e.g. pre- and postconditions. Additionally, it generates verification annotations for memory safety: in particular permissions to specify ownership of heap elements, like objects and their fields. The reasoning happens on the level of the choreography, and is preserved in the generated implementation [15]. VerCors [1] is used as the back-end verification engine for VeyMont.

An example VeyMont choreography is given in Fig. 1a. It defines two endpoints `a` and `b` of class `Role` (lines 2 and 3). The class `Role` has a field `x` of type `int`. The `run` declaration defines actions (line 8): `a` increments `a.x`, and then sends the value stored in `a.x` to `b`. Then `b` increments `b.x`, and sends it back to `a`. The precondition of `run` (line 5) is that `a.x` is more than 0. This is an example of a constraint on input data, and necessary to prove the postconditions: that `a.x` and `b.x` are equal (line 6), and that the value of `a.x` is more than 2 (line 7).

In this paper we extend VeyMont to address two limitations of the original implementation. The first limitation is the single-owner policy, where each endpoint owns all the fields reachable from it. For example, in Fig. 1a, `a` owns its only field `a.x`, while `b` owns `b.x`, in any program state, and sharing is only supported via duplicated values. This choice allowed to automatically generate permission annotations for verifying memory safety with the VerCors back-end.

Unfortunately, the single-owner policy excludes concurrent programs that share memory between threads. This is problematic, because sharing memory is an important pattern in many concurrent programs. Also, choreographies with a single-owner policy do not scale well for more endpoints, and for large data structures the overhead of duplication is large. Finally, the single-owner policy disallows sharing read-only data structures. What we instead wish to have is shared memory as used in Fig. 1b. The choreography is the same as Fig. 1a, except that endpoint `a` is initialized with a reference to a store `s`, which is used to update the field `x` within. This reference is then communicated between `a` and `b`, instead of the literal integer. While the choreography still includes `communicate` statements to prevent data races, access to `x` is now shared. In this paper, we extend VeyMont with transfer of ownership using `communicate` statements, i.e. at line 10 of Fig. 1b, ownership of `s` should be transferred from `a` to `b`. It is safe to transfer ownership here, because the receiver waits for the message of the sender. This implied synchronization points justifies permission transfers.

A second limitation of the original VeyMont [7] is that the verification annotations of a choreography are not preserved in the generated implementation. Consequently, the verification properties cannot be directly verified on the implementation. Since the properties have not been proven to hold for any variant of the program, an adapted and verified implementation can only be obtained by adapting and verifying the choreography, and then generating the implementation. From an engineering viewpoint, and especially for small changes, this may cause unnecessary overhead. Also, it is risky to change the generated implementation, e.g. for performance, as this might introduce bugs that cannot be detected by verification. Additionally, the lack of annotations in the generated implementation also prevents application of tools that further process & leverage annotations. One example of such a tool is Alpinist [22], which is a GPU program optimizer that uses annotations to check applicability of optimizations and preserves annotations in the output program.

Furthermore, because a generated implementation without annotations cannot be verified, bugs in the code generator of VeyMont are not spotted. In this work, we increase confidence in the generated implementation by making it verifiable with VerCors. This allows the user to establish correctness of the generated implementation without depending on the implementation details of VeyMont.

Contributions. In this work we present an extension of VeyMont that supports choreographies with shared memory and preserves verification annotations in the generated implementations. VeyMont supports fine-grained and dynamic ownership via endpoint annotations. In particular, access to shared memory can be exchanged between endpoints by annotating a `communicate` statement with permissions. By extending this approach to expressions and statements, VeyMont can generate implementations with verification annotations. We demonstrate the extended VeyMont through three consecutive improvements of the Tic-Tac-Toe case study, as presented in the original VeyMont tool paper [7]. These case studies show that even in simple programs, complicated properties can emerge. We provide the full annotated programs and tool implementation in the artifact [21].

Paper Structure. After preliminaries (Section 2), we describe the workflow and choreographic language of VeyMont, and introduce our approach with an example (Section 3). Then, we elaborate how choreographies are verified (Section 4), and how the concurrent program with verification annotations is generated (Section 5). After, we discuss our case studies (Section 6), and related work (Section 7). Finally, we conclude and discuss future work (Section 8).

2 Deductive Verification with VerCors

We introduce Prototypical Verification Language (PVL), the internal language of VerCors, because we used it to define the choreographic language of VeyMont. Also we introduce permission-based separation logic (PBSL), which is part of PVL, and is leveraged to verify choreographies. VerCors can also verify other languages, such as Java and C. We refer to [1] for more information about VerCors.

PVL is an object-oriented language. It supports OOP concepts like classes, methods and fields. PVL programs are imperative, with mutable local variables, `if` and `while` statements. Additionally, PVL programs may have contracts and assertions. Method contracts have pre- and postconditions, specified using `requires` and `ensures`. Assertions on expressions can be placed in a method body using the `assert` keyword. For examples, see lines 5, 6 and 11 of Fig. 4. More information on PVL is available at [1,25].

Permission-based Separation Logic. VerCors is able to verify functional correctness, e.g. via pre- and postconditions and loop invariants. It also verifies memory safety and data-race freedom using permission-based separation logic [6]. Permissions specify which fields are writable or readable using the following syntax: `Perm(`$o.x$`, `f`)`. Here, o is an object, x a field, and f a fractional permission amount $0 < f \leq 1$. Fraction $f = 1$ specifies read and write access. Fraction $0 < f < 1$ specifies only read access. Fields that are not specified with `Perm` are inaccessible. Permissions can be combined in an expression using the separating conjunction `**` operator, such that the sum of fractions never exceeds 1 for a field. Permissions can be split and combined, e.g. `Perm(o.f, 1)` can be split into `Perm(o.f, 1\2) ** Perm(o.f, 1\2)`. Note the use of "`\`" to indicate fractional division, as opposed to integer division using "`/`". Expressions in PBSL are "self-framing", which means that an expression must specify a permission for a field before reading it. For example, `Perm(x.f, 1) ** x.f == 0` is self-framing.

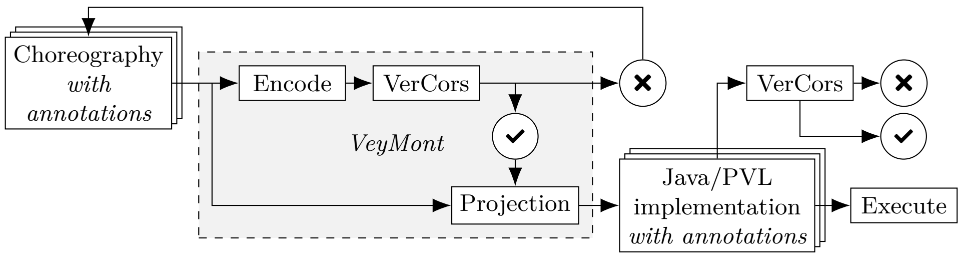

Fig. 2: Workflow for using VeyMont choreographies

VerCors also supports predicates, which allow grouping expressions, including permissions, under an opaque name. They are defined using the syntax: `resource` $P(\overline{Ta})$ `=` A, where P is the predicate name, $\overline{Ta}$ is a sequence of typed parameters, and A is the predicate body. As predicates are opaque, VerCors requires annotations that specify when a predicate body should be exchanged for a predicate name and its arguments, and vice versa. This is done with the `fold` and `unfold` statements. More specifically, if the expression A holds in the current verification state, then `fold` $P(\overline{e})$ will cause all permissions in A to be removed from the verification state. Then, the predicate $P(\overline{e})$ is added to the verification state. The `unfold` statement does the inverse: it removes $P(\overline{e})$, and adds A. See for example lines 7 and 8 of Fig. 5b. There is also `\unfolding`, which is for use in expressions. It unfolds a predicate temporarily while evaluating an expression, and folds it directly afterwards.

To mediate access to shared resources, PVL supports lock invariants, which defines assertions only available by acquiring a lock [11]. A lock invariant is declared on a class. By locking an object, the lock invariant is acquired, after which the lock invariant of the object is added to the current verification state. When unlocking the object, the lock invariant needs to be true in the current verification state, and if so its assertions are removed from the state. A lock invariant is declared at lines 1 to 5 of Fig. 8b.

3 VeyMont Workflow and Choreography Language

We now discuss the workflow and choreography language of VeyMont, including the new features that we added to support shared memory.

3.1 VeyMont Workflow

The workflow for using VeyMont is shown in Fig. 2, and consists of 3 steps:

Step 1: Verify. When a choreography with annotations is input into VeyMont, the semantics of the choreography is encoded (see Section 4), such that VerCors can verify it. If verification fails, there is a problem with the choreography: either it has a bug, or the contracts are not properly specified.

```
⟨GlobalDecl⟩ ::= ⟨Class⟩                          ⟨ChorStmt⟩ ::= · · ·
    | ⟨Procedure⟩                                     | [⟨Name⟩:] ⟨Expr⟩ . ⟨Name⟩ := ⟨Expr⟩;
    | [⟨Contract⟩]                                    | channel_invariant ⟨Expr⟩;
      choreography ⟨Name⟩ (⟨Params⟩) {                 communicate [⟨Name⟩:] ⟨Expr⟩ -> [⟨Name⟩:] ⟨Expr⟩;
        ⟨ChorDecl⟩*
      }                                           ⟨Expr⟩ ::= · · ·
                                                      | Perm[⟨Name⟩](⟨Expr⟩, ⟨Expr⟩)
⟨ChorDecl⟩ ::=                                        | (\endpoint ⟨Name⟩; ⟨Expr⟩)
    | endpoint ⟨Name⟩ = ⟨Name⟩(⟨Args⟩);               | (\chor ⟨Expr⟩)
    | [⟨Contract⟩] run { ⟨ChorStmt⟩* }                | \msg
    | ⟨Method⟩                                        | \receiver
                                                      | \sender
```

Fig. 3: VeyMont syntax in EBNF.

Step 2: Endpoint Projection. If verification succeeds, VeyMont applies endpoint projection on the choreography to generate an implementation for each choreography endpoint, with annotations. Section 5 discusses the endpoint projection. VeyMont can generate both PVL and Java code.

Step 3: Use. The generated implementation can be used in two ways. First, if Java code was generated, it can be executed. Second, for PVL code, it can be verified with standalone VerCors. If standalone verification fails, there was either a bug in the projection step, the choreography contains annotations that cannot be projected (using `\chor`, see Section 3.2), or the user made changes to the generated implementation that are incompatible with the current annotations.

3.2 Choreography Language

VeyMont extends the PVL language with syntax for defining choreographies. This syntax extension is summarized informally in Fig. 3.

Global Declarations. VeyMont definitions coexist with other PVL definitions such as classes. This way VeyMont programs can use procedures and types from PVL. VeyMont adds a type of global declaration, the `choreography`. This is the root definition for a VeyMont choreography. It consists of an optional contract, a name, parameters, and zero or more choreography declarations. In a VeyMont program, multiple choreographies can co-exist simultaneously.

Choreographic Declarations. A choreographic declaration is either an `endpoint` declaration, a `run` declaration, or a method declaration. An `endpoint` declaration defines an endpoint that participates in the choreography. Semantically this corresponds to an object created with a constructor. An endpoint has a name, a name of a PVL class, and a list of expressions as arguments for the constructor. The `run` declaration consists of an optional contract, and a body consisting of choreographic statements. Lastly, a regular PVL method definition is also a *ChorDecl*, when its body consists of choreographic statements.

Choreographic Statements. There are two choreographic statements: choreographic assignment and the `communicate` statement. Choreographic assignment

(`:=`) is similar to regular assignment, with the restriction that the value can only be computed using state from one endpoint. It consist of an expression of the object being assigned to, the target field, and an expression. Moreover, the choreographic assignment can optionally be labeled with an endpoint name to enable using a shared field from a different endpoint. The following verifies if `a` has permission for both `b.f` and `a.f`: "`a: b.f := a.f;`".

The `communicate` statement sends a value from one endpoint to another. It requires a receiving endpoint and field, and a sending endpoint and expression. When endpoints are omitted, they are derived from the expressions, e.g. `a.x` has `a` as the implicit endpoint. `communicate` statements are semi-synchronous: sending is non-blocking, receiving is blocking. Annotations for ownership transfer of a `communicate` statement, as well as functional constraints over the message, can be specified in a `channel_invariant` annotation. For example, the annotation "`channel_invariant \msg > 2; communicate a.x -> b.x;`" can be added to verify that the message sent from `a.x` to `b.x` is bigger than 2. Currently, only the choreographic expressions (explained in the next paragraph) `\msg`, `\sender`, `\receiver` as well as global functions, are allowed within `channel_invariant`. This is purely a limitation of the current implementation, as channels can straightforwardly be extended with extra context.

Within a `choreography` declaration, also selected PVL statements are allowed, such as `assert`, `assume`, `if`, `while` and blocks of statements.

Choreographic Expressions. There are six choreographic expressions. An endpoint name can be denoted within brackets at a permission annotation, to specify ownership by the endpoint of the stated permission. The keyword `\endpoint` requires the name of an endpoint and an expression. This indicates that, in the encoding (Section 4), the expression should only be evaluated for the endpoint, and only included in the implementation of the endpoint (Section 5). Within a `channel_invariant`, three additional expressions are available. These are `\msg`, `\sender` and `\receiver`. They are used to indirectly refer to the sender, receiver, and message of the next `communicate` statement.

Finally, the `\chor` keyword wraps an expression. This expression can access memory of all endpoints in the choreography. Specifically, within `\chor` *endpoint ownership annotations* are ignored. Because of this, it cannot be included in the generated implementation. Consequently, if a choreography contains a `\chor` expression, the generated implementation might not verify. More formally, we believe that if a correct choreography does not contain `\chor`, the generated implementation also verifies. We have not proven this, and leave it for future work. The `\chor` keyword is included because it makes the workflow of the original VeyMont [7] possible. It is also useful for prototyping contracts of a choreography, as endpoint ownership annotations prevent asserting expressions that ignore endpoint ownership, which limits the user when debugging annotations. Finally, `\chor` serves a similar role as the `assume` statement. Once a choreography is proven correct, the `\chor` should be removed. An example of `\chor` is in the TTT case study on page 14.

```
 1  choreography increment(Store s) {
 2    endpoint a = Role(s);
 3    endpoint b = Role(null);
 4
 5    requires Perm[a](a.s, 1\2) ** Perm[a](a.s.x, 1) ** a.s.x > 0;
 6    ensures Perm[a](a.s, 1\2) ** Perm[a](a.s.x, 1) ** a.s.x > 2;
 7    run {
 8      a: a.s.x := a.s.x + 1;
 9      channel_invariant Perm(a.s, 1\2) ** Perm(a.s.x, 1) ** a.s.x > 1;
10      communicate a: a.s -> b: b.s;
11      assert Perm[b](a.s, 1\2) ** Perm[b](a.s.x, 1);
12      b: a.s.x := a.s.x + 1;
13      channel_invariant Perm(a.s, 1\2) ** Perm(a.s.x, 1\2) ** a.s.x > 2;
14      communicate b: b.s -> a: a.s;
15    } }
```

Fig. 4: A choreography where the endpoints increment a shared field with ownership annotations. Adapted from Fig. 1b. Note that some endpoint ownership annotations that can be inferred by VeyMont are included for clarity.

3.3 New Features of VeyMont

This paper introduces two extensions for VeyMont choreographies: endpoint ownership annotations, and channel invariants.

Endpoint ownership annotations indicate the owner of a permission. When an endpoint e owns a permission `Perm(`$o.x$`,` f`)`, this is written as `Perm[`e`](`$o.x$`,` f`)`. For example, the permission `Perm[alex](`$o.x$`, 1)` allows `alex` to write to field x of object o. When a user writes `Perm(alex.x, 1)`, VeyMont infers automatically that `Perm[alex](alex.x, 1)` was meant. By explicitly writing `Perm[bob](alex.x, 1)` the user specifies that `bob` currently has writing access to `alex.x`, while `alex` has no access. This way, `alex.x` is used as shared memory. Assignments can be annotated similarly: e`:` $o.f$ `:=` v denotes that endpoint e executes assignment $o.f$ `:=` v. Again we allow shorthand notation: `alex.x := 1` denotes `alex: alex.x := 1`. Communications are written as `communicate` s`:` v `->` r`:` u, where endpoint s sends value v to receiver r, which stores it in u. The shorthand notation `communicate alex.x -> bob.y` is also supported.

Channel invariants allow access to memory to be exchanged, i.e. shared. This is done by adding a `channel_invariant` annotation on a `communicate` statement. E.g. `channel_invariant Perm(alex.x, 1)` gives the receiver write access to `alex.x`, while the sender has lost access after this communication. In other words, permissions are transferred between the sending and receiving party *if and only if* they are stated in the channel invariant.

Motivating Example We will now further demonstrate these concepts using an example. Figure 4 shows how we annotate the program from Fig. 1b so that VeyMont can verify the program with the shared field.

Endpoints `a` and `b` are initialized at lines 2 and 3. Then, line 5 states the precondition of the `run` method: `a` has write access to `a.s.x`. On line 8, `a` in-

```
 1 choreography setter(Store s) {
 2   endpoint a = Role();
 3   endpoint b = Role();
 4
 5   requires Perm[a](s.x, 1);
 6   run {
 7     a: s.x := 0;
 8     b: s.x := 1;
 9   }
10 }
```

(a) An incorrect choreography

```
 1 resource perm_x(Role e, Store s) = Perm(s.x, 1);
 2
 3 requires perm_x(a, s);
 4 void setter_run_a(Store s, Role a, Role b) {
 5   unfold perm_x(a, s);
 6   s.x = 0;
 7   fold perm_x(a, s);
 8   unfold perm_x(b, s);
 9   s.x = 1;
10   fold perm_x(b, s);    }
```

(b) Encoding of `run`

Fig. 5: A choreography and encoding of `run` using permission stratification.

crements `a.s.x`. This is explicitly denoted with `a:` at the start of the line. On lines 9 and 10, the reference to `Store s` is sent from `a` to `b`. In addition, the channel invariant transfers write access for `a.s.x` from `a` to `b`. This is explicitly verified with the `assert` on line 11. On line 12, `b` performs an increment to `a.s.x`, and then proceeds with the `communicate` statement on line 14, to send write permission back to `a`. The postcondition on line 6 states these write permissions, and additionally that `a.s.x` is more than 2. VeyMont will verify the program, in particular that the postcondition will hold. After that, VeyMont can also be invoked to generate the corresponding concurrent program with threads for endpoints `a` and `b`.

4 Choreography Verification

VeyMont generates an encoding of a choreography such that VerCors can verify it. This encoding essentially collapses all endpoint behaviors into one implementation. To prevent permissions of different endpoints from being accidentally combined, permissions are stratified (Section 4.1). This also allows encoding the transfer of the message and permissions between two endpoints (Section 4.2).

4.1 Permission Stratification

To encode permissions labeled with an endpoint owner into PVL, we use PVL predicates to label a permission with its endpoint owner. For each permission annotated with an endpoint owner, we create a predicate wrapping that permission. To this predicate we add an argument that models the endpoint owner. In essence, this argument enforces that a predicate can only be unwrapped if the current endpoint owner is specified. We call this technique "permission stratification". For example, the permission on line 5 of Fig. 5a results in the predicate on line 1 in Fig. 5b, where `[a]` in the annotation causes creation of the argument `e` in the predicate. The argument `e` is only used to distinguish stratified permissions with different owners, and therefore does not occur in the predicate.

Adding an extra argument to the predicate, to encode which endpoint owns the permission, works because of the following: unfolding a predicate only succeeds if the correct arguments are used. In this case, unfolding means exchanging a predicate instance for its body, which in turn modifies the verification state. For example, on line 3 in Fig. 5b, the predicate `perm_x(a, s)` is required. Within the method, the permission within the predicate can only be accessed using `unfold perm_x(a, s)`. Conversely, the statement `unfold perm_x(b, s)` would fail, as there is only a predicate `perm_x(a, s)` present. Because the arguments have to be stated explicitly to unfold the predicate, the extra argument effectively acts as a "key" to access the permission within the predicate.

VeyMont unfolds wrapper predicates automatically when the endpoint owner of a permission is known, for example, in the case of assignment. This makes the permission in the predicate available for verification. Later, VeyMont folds the predicate, possibly with a new endpoint owner. The `fold` and `unfold` steps are generated by VeyMont according to inferred or user-supplied annotations, and are checked by VerCors. For example, VeyMont generates `unfold` and `fold` annotations before and after assignment to fields. This is shown on lines 5 and 7 of Fig. 5b.

The example in Fig. 5a shows an incorrect choreography. The two endpoints share a `Store s` and each writes to it. The user specifies that `a` owns the store on line 5. This program contains a data race: `a` and `b` run concurrently and write to the same location. Therefore, verification will fail, with an error on line 8.

The example in Fig. 5b shows the encoded choreography with all permissions stratified. Line 1 defines a wrapping predicate for when the field `x` is owned by a given endpoint `Role e`. Line 4 encodes the choreography parameter `Store s`, and endpoints `a` and `b`. Verification with VerCors yields that on line 8 it cannot unfold predicate `perm_x(b, s)` because it is not present in the verification state. Indeed, after line 7, the verification state holds exactly `perm_x(a, s)`, and not `perm_x(b, s)`! One way to fix this example is to send the permission `Perm(s.x, 1)` from `a` to `b` between the two assignment statements using a `communicate` statement. This will exchange `perm_x(a, s)` with `perm_x(b, s)`, at the cost of synchronization at run-time (see Section 4.2).

By wrapping permissions in predicates, VeyMont can verify the behavior of multiple endpoints within one program. This is the key ingredient that allows verification of choreographies with shared memory.

4.2 Encoding of Choreographic Communication

Figures 6b and 6c show how VeyMont encodes the `communicate` statement and `channel_invariant` of Fig. 6a. This is an example to illustrate the encoding, it is not meaningful on its own. All line numbers in this subsection refer to Fig. 6c. Summarizing, the encoding consists of 4 steps: line 6 encodes message evaluation, line 9 encodes channel invariant checking, lines 12 and 13 encode the transfer of the channel invariant from `a` to `b`, and lines 15 to 17 encode message reception. The `fold` annotations are required for handling stratified permissions, following the explanation in the previous section.

```
choreography incr(int i) {
  endpoint a = Role(i); endpoint b = Role(i);
  requires Perm(a.x, 1) **
    (c() ==> Perm(a.z, 1) ** a.z == a.x);
  requires Perm(b.y, 1);
  run {
    channel_invariant
      c() ==> Perm(a.z, 1) ** \msg == a.z;
    communicate a.x -> b.y;
  } }
```

(a) Input choreography

```
// For each field f ∈ {c, x, y, z}, define:
resource perm_f(Role e, Role r) = Perm(r.f, 1);
int get_f(Role e, Role r) =
  (\unfolding perm_f(e, r) \in r.f)
```

(b) Background definitions for encoding

```
1  requires perm_x(a, a) **
2    (c() ==> perm_z(a, a) **
3      get_z(a, a) == get_x(a, a);
4  requires perm_y(b, b);
5  void incr_run(int i, Role a, Role b) {
6    // Evaluate message
7    int m = get_x(a, a);
8    // Assert invariant
9    assert (c() ==> perm_z(a, a) **
10     m == get_z(a, a));
11   // Transfer invariant
12   if (c()) { unfold perm_z(a, a); }
13   if (c()) { fold perm_z(b, a); }
14   // Store message at target
15   unfold perm_y(b, b);
16   b.y = m;
17   fold perm_y(b, b);
18 }
```

(c) Encoding of choreography

Fig. 6: Encoding of a choreography with a `channel_invariant` and `communicate` statement. For brevity, method definition `incr` has been omitted in Fig. 6c.

First, the message to be sent is stored in `m` on line 6. To read `a.x`, the function `get_x` is used. Each `get_f` function unfolds the wrapper predicate `perm_f` to read field *f*. On line 9, the channel invariant is checked using `assert`. Note that the channel invariant was transformed: `m` replaces `\msg`, and a wrapper predicate replaces `Perm(a.z, 1)`, following the stratified permissions approach. The owner of this wrapper predicate is `a`, because `a` is the sender of the communication.

The channel invariant is transferred from `a` to `b` on lines 12 and 13 via the `unfold` and `fold` statements. The `channel_invariant` contains the condition `c()`, which is an abstract global condition defined for this example. Because of this condition, `if` statements are also necessary to conditionally `unfold` and `fold` the predicates that wrap permissions. For the boolean parts of the invariant, no annotations have to be added, as these are kept track of automatically by the symbolic execution back-end of VerCors. Finally, `m` is assigned to target location `b.y` on line 16, which models the receiving of the message by `b`.

5 Endpoint Projection

To generate an implementation for an endpoint of a given choreography, the endpoint projection translates each statement depending on which endpoint is currently the target. We extend the endpoint projection presented in [7] to take into account endpoint ownership annotations. This allows VeyMont to include contracts in the projection, making the generated implementation verifiable if correct annotations are provided. In addition, we show how channel invariants are included in the channel classes that implement `communicate` statements.

Choreography with `a` & `b`		Projection for: `a`	Projection for: `b`
`a.x := 5;`	→	`a.x = 5;`	`/* skip */`
`communicate a.x -> b.y;`	→	`a_b.writeValue(a.x);`	`b.y = a_b.readValue();`
`if (a.x == 5 &&`	→	`if (a.x == 5 &&`	`if (true &&`
`b.y == 9) {`	→	`true) {`	`b.y == 9) {`
`a.foo(a.x); }`	→	`a.foo(a.x); }`	`/* skip */ }`
`b: a.x := 5;`	→	`/* skip */`	`a.x = 5;`
`communicate b: a.x -> a: b.y;`	→	`b.y = a_b.readValue();`	`a_b.writeValue(a.x);`
`Perm[a](x.f, 1)`	→	`Perm(x.f, 1)`	`true`
`(\chor` v`)`	→	`true`	`true`
`(\endpoint` a`;` v`)`	→	v	`true`

Fig. 7: Summary of endpoint projection rules by example. Top half describes rules from [7]. Bottom half describes endpoint ownership annotations.

5.1 Statement & Expression Projection Rules

Figure 7 shows – by example – the endpoint projection rules to generate an implementation for a target endpoint. The top half of the table shows the rules identical to those in [7], the bottom half shows the new rules. Using these new rules, contracts and loop invariants can straightforwardly be transformed and preserved in the generated implementation, which was previously not possible.

We will now further discuss the rules in the bottom half of Fig. 7. If the target endpoint participates, i.e. occurs, in a statement or expression, it is transformed as follows: choreographic assignment (`:=`) is transformed into plain assignment, `communicate` statements are transformed into invocations of `readValue` and `writeValue` methods on channel objects. `Perm[`e`](`$o.x$`,` f`)` is included without `[`e`]` in the generated implementation, and similarly, the keyword `(\endpoint` e`;` v`)` causes v to be included in the generated implementation. If the target endpoint does not participate in a statement or expression, it is omitted or replaced with `true`. The keyword `(\chor` v`)` is handled by always discarding it. This is because `\chor` can freely access the memory of all endpoints, and hence cannot safely be included in the generated implementation (see Section 3.2).

5.2 Generating Channels

For each `communicate` statement (Fig. 8a) VeyMont generates a distinct channel class (Fig. 8b). An instance of this class is constructed at the start of the program, and both endpoints of the `communicate` statement are given a reference to it. To send and receive values, the methods `writeValue` and `readValue` can be called. The `lock_invariant` expresses that the `channel_invariant` holds at the moment of transfer, i.e. when `writeValue` has written the communicated value in `msg` and set `hasMsg` to true, and `readValue` has been called after that. Because `writeValue` has the `channel_invariant` as precondition, and `readValue` as postcondition, the transfer of the `channel_invariant` is achieved.

```
channel_invariant
  Perm(\sender.z, 1) **
  Perm(\receiver.z, 1) **
  \sender.z == \receiver.z;
communicate a.x -> b.y;
```

(a) `communicate` with corresponding `channel_invariant`

```
lock_invariant
  Perm(hasMsg, 1) ** Perm(msg, 1) **
  Perm(s, 1\2) ** Perm(r, 1\2) **
  (hasMsg ==> Perm(s.z, 1) **
    Perm(r.z, 1) ** s.z == r.z);
class ChanAB {
  boolean hasMsg; int msg;
  Role s, r; // Sender, receiver

  context Perm(s, 1\8) ** Perm(r, 1\8);
  requires Perm(s.z, 1) ** Perm(r.z, 1) **
    s.z == r.z;
  void writeValue(int m);

  context Perm(s, 1\8) ** Perm(r, 1\8);
  ensures Perm(s.z, 1) ** Perm(r.z, 1) **
    s.z == r.z;
  int readValue(); }
```

(b) Generated channel class.

Fig. 8: Generated channel class for `channel_invariant` and `communicate`.

Since a channel invariant may refer to both `\sender` and `\receiver`, the generated class contains both endpoints as references `s` and `r`. Read permissions for these fields reside at both endpoints. This way, respective fields of `\sender` and `\receiver`, e.g. `\sender.z`, can be expressed as `s.z`, when calling `writeValue` or `readValue`. The omitted implementations of `writeValue` and `readValue` are standard, where `writeValue` does not block, but `readValue` does.

6 Case Studies

To demonstrate the VeyMont extension of this paper, we present case studies on three variants of Tic-Tac-Toe. Here TTT is the baseline case study, adapted from [7], TTT$_{\text{msg}}$ uses ownership annotations, and TTT$_{\text{last}}$ optimizes away a theoretically unnecessary message. We provide the full annotated programs and tool implementation in the artifact [21].

The TTT case study is a variant of the case study discussed in [7]. It is set up to simulate a game of tic-tac-toe on a 3×3 in a distributed setting. This means each endpoint has its own local copy of the board, and as the endpoints take turns they send their moves to each other so the boards stay in sync. When a winning move occurs, or the board runs out of spaces, the game ends.

While each case study has different annotations, the postcondition proven is the same: *after the game terminates, the boards of the two players are identical.*

This postcondition highlights the complexity of verifying an easy to understand choreography. To prove correctness, VeyMont must prove that each move made by one player is also applied to the local board of the other player. This kind of property could also occur when e.g. executing a transaction in a distributed database. When verifying the TTT choreography and ignoring permission stratification, the property is proved automatically. However, once the endpoints are

```
1  choreography TTT() {
2    endpoint p1 = Player(0, true);
3    endpoint p2 = Player(1, false);
4
5    requires p1.myMark == 0 **
6      p2.myMark == 1 **
7      (\chor p1.turn != p2.turn **
8        p1.equalBoard(p2));
9    ensures (\chor p1.equalBoard(p2));
10   run {
11     loop_invariant /* omitted */ **
12       p1.equalBoard(p2);
13     while(!p1.done() && !p2.done()) {
14       if(p1.turn && !p2.turn) {
15         p1.createNewMove();
16         communicate p1.move.copy() -> p2.move;
17       } else {
18         p2.createNewMove();
19         communicate p2.move.copy() -> p1.move;
20       }
21       p1.doMove();
22       p2.doMove();
23       p1.turn := !p1.turn;
24       p2.turn := !p2.turn;
25     }
26   }
27 }
```

Fig. 9: Main choreography of the TTT case study. The loop invariant on line 11 is omitted as it is the same as the precondition of `run`.

split up into threads with the endpoint projection, a problem arises: the property becomes impossible to state. This is because the property requires player one to make an assertion about the state of player two, and vice versa.

Each case study solves this problem differently. The TTT case study solves the problem by using `\chor`. This allows violating the restriction of stratified permissions, at the cost of missing annotations in the generated implementation. Case studies $\mathrm{TTT}_{\mathrm{msg}}$ and $\mathrm{TTT}_{\mathrm{last}}$ use stratified permissions to pass permissions back and forth, ensuring the players can alternate reading and writing to both boards safely. Specifically, $\mathrm{TTT}_{\mathrm{msg}}$ introduces an extra message at run-time, and $\mathrm{TTT}_{\mathrm{last}}$ eliminates this run-time overhead by using more complicated annotations. For $\mathrm{TTT}_{\mathrm{msg}}$ and $\mathrm{TTT}_{\mathrm{last}}$, the generated implementations *do* verify.

TTT. The TTT case study is similar to the case study presented in [7]. The only changes are the reduction to a 3×3 board instead of an $M \times N$ board, and minor syntactical changes. This is not a limitation of this is paper, it is merely a simplification for ease of presentation. The choreography of TTT is shown in Fig. 9. After the endpoints are initialized, the endpoints enter a loop, where they alternate taking turns. After each turn, the move is send to the other player so they can update their board. The postcondition is `(\chor p1.equalBoard(p2))`, meaning the board of `p1` is equal to that of `p2` after termination. With automatic permission generation enabled, VeyMont can verify the choreography with the initial approach presented in [7]. The projection on these (old style) choreographies yields generated implementations where the choreography properties hold [15], but verification annotations marked with `\chor` are not present in the generated implementations, and hence the choreography postcondition cannot be verified on it.

TTT_{msg}. We take a different approach to avoid `\chor`: each endpoint will only keep half permission for their own board. The other halves are pooled and used to establish and maintain board equality. After each turn, these pooled permissions

```
1  channel_invariant
2    \msg.movePerm() **
3    ([1\2]\sender.boardPerm()) **
4    ([1\2]\receiver.boardPerm()) **
5    \sender.oneMoveAheadOf(\msg, \receiver);
```

(a) Channel invariant of the communications in `while` loop

```
if (p1.turn && !p2.turn) {
  channel_invariant
   [1\4]\sender.boardPerm() **
   [1\4]\receiver.boardPerm() **
   \receiver.equalBoard(\sender);
  communicate p1: true -> p2;
}
```

(b) Communication at game end

Fig. 10: Communications in TTT$_{\text{msg}}$ case study.

are sent to the other player. Finally, when the game ends, the last player splits the pooled permissions sends half to the other player. This gives both players read permission to both boards, allowing them to state board equality.

To this end, we add to both communications in the while loop the channel invariant of Fig. 10a. The sending player provides `1\2` permission for his own board, *and* the other players board, in the channel invariant (using prefix scaling notation `[1\2]` before the predicate on lines 3 and 4). This invariant implies that the sending player is exactly one move ahead (line 5). This makes sense as the receiving player still has to update the board with the communicated move after the communication. Each player always keeps `1\2` permission for his own board.

At each point in the game, only one of the players can read both players' boards, thus only one player is able to verify that the boards are equal. When the game ends, one of the players sends `1\4` permission for both boards to the other player. Figure 10b shows this for `p1`. This way, postcondition `p1.equalBoard(p2)` can be stated by both endpoints without `\chor`, and hence proven directly for the whole generated concurrent program.

TTT_{last}. We optimize away the `communicate` statement after the while loop, while still retaining correctness. We do this by introducing additional ghost state and reformulating the annotations to be more general. In doing so, we demonstrate a trade-off: run-time overhead can often be eliminated, at the cost of additional complexity in the contracts and ghost state.

Specifically, we use extra fields `p1.lastPlayer` and `p2.lastPlayer`. These store a reference to the same object, which stores the mark of the "last player". In Fig. 11b we see that, just before communicating a move, `p1` checks if this move ended the game. If so, `p1.lastPlayer` is set to `p2.myMark`, because `p2` will be the last player updating his board, before the game ends. The predicate `p1.lastPlayerPerms()` specifies write permission to the `mark` field of its `lastPlayer` object. If the game is not finished yet, `p1` includes the full permission (`1\2` + `1\2`) in the channel invariant, so that `p2` may (possibly) edit it. Otherwise, if the game is finished, only `1\2` permission is sent, such that both players can read their `lastPlayer.mark` field to see who was the last player. In Fig. 11a we show the adapted postcondition of `run`: the player whose mark is stored in its `lastPlayer.mark` field can ensure `p1.equalBoard(p2)`. This

```
ensures (\endpoint p1;
  ([1\2]p1.lastPlayerPerms() **
  p1.lastPlayer.mark == p1.myMark ==>
    ([1\2]p1.boardPerm() **
    ([1\2]p2.boardPerm()) **
    p1.equalBoard(p2));
```

(a) Postcondition of `run` method.

```
if (p1.gameFinished()) {
  p1.lastPlayer.mark = 1 - p1.myMark;
}
channel_invariant /* ... */
  ([1\2]\sender.lastPlayerPerms() **
  (!\sender.gameFinished() ==>
   ([1\2]\sender.lastPlayerPerms()));
communicate p1.move.copy() -> p2.move;
```

(b) Code for marking the last player, and the channel invariant extension.

Fig. 11: Adapted code of TTT_{last} with respect to TTT_{msg}, stated for `p1`. It is symmetric for `p2`.

postcondition ensures that when both endpoints terminate in the generated implementation, VeyMont will conclude that whichever endpoint is the last player, there will always be one of them that will guarantee board equality. In this way, `p1.equalBoard(p2)` can be proven directly for the whole generated program.

7 Related Work

Besides the works we build upon by extending VeyMont [7,15], the most similar research in the realm of VerCors is the work by Darabi et al. [2]. They introduce the `send` and `receive` statements to model loop dependencies. These statements allow sending permission to other iterations of a loop, and are similar to `communicate`. However, these statements are only supported inside loops, offer no support for sending a value, and conditional sends can only depend on variables not modified inside the loop.

Similar works in the area of choreographies are on logics to reason about the correctness of choreographies [9,10]. These works could serve as a basis for formalizing the approach outlined in this paper, but they would have to be extended with support for separation logic.

We see interesting correspondences with multiple works on session types. Generally, session types do not support implementation generation. In theory, session type results may be transferable to choreographies, but this step is nontrivial.

Hinrichsen et al. [12] introduce Actris, a Coq framework using Iris for correctness reasoning over session types. Jacobs et al. [14] introduce similar but smaller formalization of dependent session protocols, also in Iris. Both approaches are powerful, but being Coq frameworks, lack the automation we aim for in this paper. They could be good starting points for formalizing our approach.

Neykova et al. [19] present SPY, a tool that generates run-time monitors of user-defined constraints on exchanged messages and endpoint state. Our approach works without running the code, and introduces no overhead at run-time.

Bouma et al. [8] use VerCors to check conformance of Java programs to a multi-party session type (MPST). Specifically, they use permissions only at the implementation level, while we already use permissions at choreography-level.

Marques et al. present an approach to verify that C programs written using MPI [17] follow a protocol defined using a session type [16]. Their tool allows constraints to be expressed over messages sent and received, which is an extended version of session types. However, the constraints are limited to (in-)equalities of arithmetic expressions and variables, while we support general first-order logic expressions. The tool also has no support for shared memory or ghost state.

Zhou et al. [27] present Session*, a tool that extends the Scribble protocol language [26] with refinement types by compiling Session* protocols to F* [23], a functional programming language with refinement types. Because mutable memory is supported within the generated callbacks implemented in F* through an effect system, Session* supports a limited form of mutability indirectly. We support it generally, allowing sharing mutable memory across implementation callbacks and reasoning about it in contracts.

Swamy et al. [24] formalize a minimal 2-party session type framework as an example use of the SteelCore separation logic framework in F* [23]. They do not offer specialized support for correctness reasoning of session types or the transfer of resources via session types, beyond what F* offers natively. We foresee that our approach could be embedded in F* using SteelCore.

Bocchi et al. [5] present a formal framework for applying design-by-contract to session types. The "global assertions" from their work are similar to contracts in VeyMont choreographies. Besides the difference between session types and choreographies, Bocchi et al. also do not support shared memory. They do define well-assertedness of global assertions to e.g. prevent endpoints from using values they do not know about. We resolve this by using permission stratification.

Finally, Proust et al. [20] have integrated the Why3 [4] program verifier with the Bulk Synchronous Parallel (BSP) model. The version of BSP in this work shares some aspects with OpenMP, as it offers parallelized versions of common operations, such as map and fold. In addition, BSP offers choreography-like many-to-many communication. There are two differences with our work. First, code written using the BSP API can only be executed in an environment that provides such an API. VeyMont generates plain Java & PVL code that can be verified and only needs the standard library. Second, Proust et al. only consider purely functional programs, while VeyMont supports reasoning about mutable variables and shared memory.

8 Conclusion

VeyMont could already verify choreographies, auto-generate permissions, and use the endpoint projection to generate an implementation. In this work, we added endpoint ownership annotations and channel invariants to VeyMont, such that choreographies can specify concurrent programs with shared memory between threads. Additionally, we transfer verification annotations to the generated implementations, such that they can be verified directly, without the choreography. We showed the new capabilities of extended VeyMont in case studies.

For future work, we first of all aim to introduce parameterized endpoints, such that distributed systems with any n number of nodes can be formulated as choreography. Also, adding support for one-to-many or many-to-one communications would make VeyMont more expressive. While we now use verification of choreographies and the generated implementations to ensure correctness of the projection, we would also like to formalize our approach, i.e. extend [15]. Finally, by doing more case studies, we will validate our approach more extensively.

References

1. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The VerCors verifier: A progress report. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14682, pp. 3–18. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_1
2. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9033, pp. 202–217. Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_14
3. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The vercors tool set: Verification of parallel and concurrent software. In: Lecture Notes in Computer Science, pp. 102–110. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-66845-1_7
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011)
5. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 162–176. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_12
6. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 259–270. ACM (2005). https://doi.org/10.1145/1040305.1040327
7. 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
8. Bouma, J., de Gouw, S., Jongmans, S.: Multiparty session typing in java, deductively. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 19–27. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_3
9. Carbone, M., Grohmann, D., Hildebrandt, T.T., López, H.A.: A logic for choreographies. In: Honda, K., Mycroft, A. (eds.) Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. EPTCS, vol. 69, pp. 29–43 (2010). https://doi.org/10.4204/EPTCS.69.3

10. Cruz-Filipe, L., Graversen, E., Montesi, F., Peressotti, M.: Reasoning about choreographic programs. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13908, pp. 144–162. Springer (2023). https://doi.org/10.1007/978-3-031-35361-1_8
11. Haack, C., Huisman, M., Hurlin, C.: Reasoning about java's reentrant locks. In: Ramalingam, G. (ed.) Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5356, pp. 171–187. Springer (2008). https://doi.org/10.1007/978-3-540-89330-1_13
12. Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris: session-type based reasoning in separation logic. Proc. ACM Program. Lang. **4**(POPL), 6:1–6:30 (2020). https://doi.org/10.1145/3371074
13. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFB0053567
14. Jacobs, J., Hinrichsen, J.K., Krebbers, R.: Dependent session protocols in separation logic from first principles (functional pearl). Proc. ACM Program. Lang. **7**(ICFP), 768–795 (2023). https://doi.org/10.1145/3607856
15. 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
16. Marques, E.R.B., Martins, F., Vasconcelos, V.T., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: Yoshida, N., Vanderbauwhede, W. (eds.) Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013. EPTCS, vol. 137, pp. 103–113 (2013). https://doi.org/10.4204/EPTCS.137.9
17. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.0 (Jun 2021), https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
18. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023). https://doi.org/10.1017/9781108981491
19. Neykova, R., Yoshida, N., Hu, R.: SPY: local verification of global protocols. In: Legay, A., Bensalem, S. (eds.) Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8174, pp. 358–363. Springer (2013). https://doi.org/10.1007/978-3-642-40787-1_25
20. Proust, O., Loulergue, F.: Verified scalable parallel computing with why3. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods - 21st

International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14323, pp. 246–262. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_14, https://doi.org/10.1007/978-3-031-47115-5_14

21. Rubbens, R., van den Bos, P., Huisman, M.: VeyMont permission annotations tic-tac-toe case studies and tool implementation (2024). https://doi.org/10.5281/zenodo.13348214
22. Sakar, Ö., Safari, M., Huisman, M., Wijs, A.: Alpinist: An annotation-aware GPU program optimizer. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 332–352. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_18
23. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655
24. Swamy, N., Rastogi, A., Fromherz, A., Merigoux, D., Ahman, D., Martínez, G.: Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. Proc. ACM Program. Lang. **4**(ICFP), 121:1–121:30 (2020). https://doi.org/10.1145/3409003
25. Vercors tool homepage. https://utwente.nl/vercors, accessed: 2024-03-01
26. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch-Lafuente, A. (eds.) Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8358, pp. 22–41. Springer (2013). https://doi.org/10.1007/978-3-319-05119-2_3
27. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 148:1–148:30 (2020). https://doi.org/10.1145/3428216