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

Robert[1] Rubbens, Petra van den Bos, Marieke Huisman

FMT Colloquium, Enschede

October 3rd, 2024

UNIVERSITY OF TWENTE.

Formal Methods & Tools

VerCors

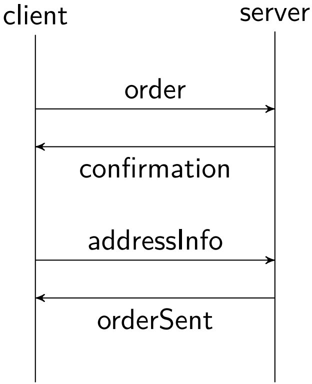
[1] (Bob)

Takeaways

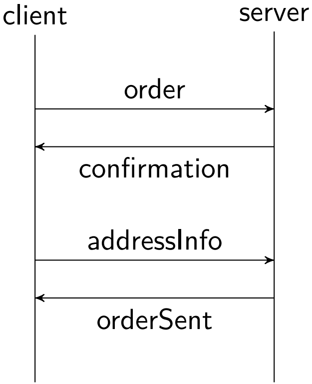
- Starting point: VeyMont: correctness by construction, single-owner memory management, verification
- Our contribution: verification as a recurring component, fine-grained memory
- How: using this one weird trick: endpoint ownership annotations
  - $\sim$ indicating per expression the relevant endpoint

Choreographies

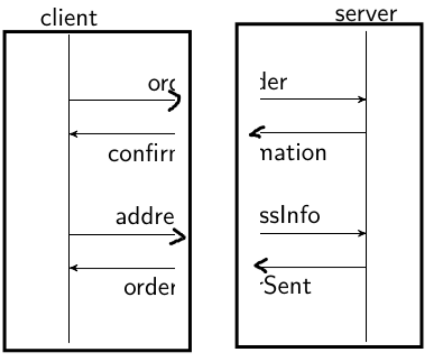
- Systems grow larger every day
  - More computers
  - More threads
- One solution: choreographies
- Choreography: series of message exchanges between 2+ parties
- Benefits:
  - Message fidelity
  - Deadlock freedom
  - Code generation



Choreographies: code generation



$\Longrightarrow$
endpoint projection



Choreographies: notation

```
choreography incr(int x) {
  endpoint alex = Role(x);
  endpoint bob  = Role(-1);

  run {
    alex.x := alex.x + 1;
    communicate alex.x -> bob.x;
    bob.x := bob.x + 1;
    communicate bob.x -> alex.x;
  }
}
```

Choreographies: notation

```
choreography incr(int x) {
  endpoint alex = Role(x);
  endpoint bob  = Role(-1);

  run {
    alex.x := alex.x + 1;
    communicate alex.x -> bob.x;
    bob.x := bob.x + 1;
    communicate bob.x -> alex.x;
  }
}
```

After executing `incr`, is `alex.x > 2`?

Functional correctness of choreographies: VeyMont

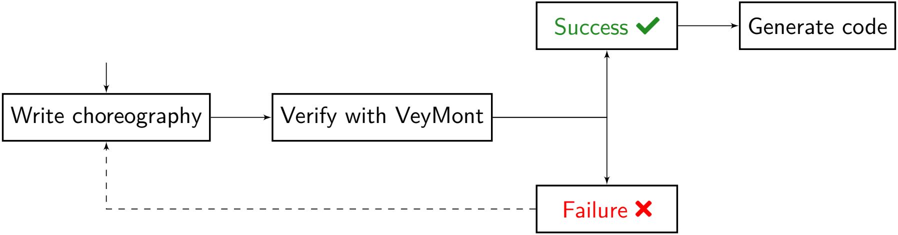
- Beyond deadlock freedom & message fidelity
- Assertions, pre- and postconditions

Metatheory [3]:

$$\text{Choreography correct} \implies \text{generated implementation correct}$$

VeyMont

- Choreography verifier & code generator [3, 2]
- Extends permission-based separation logic verifier VerCors [1]
- Choreographies look kind-of sequential, sometimes easier
- “Parallelise verified programs”



VeyMont annotations example

```
choreography incr(int x) {
  endpoint alex = Role(x);
  endpoint bob  = Role(-1);

  requires alex.x > 0;
  ensures alex.x > 2;
  run {
    alex.x := alex.x + 1;
    communicate alex.x -> bob.x;
    bob.x := bob.x + 1;
    communicate bob.x -> alex.x;
  }
}
```

Limitations

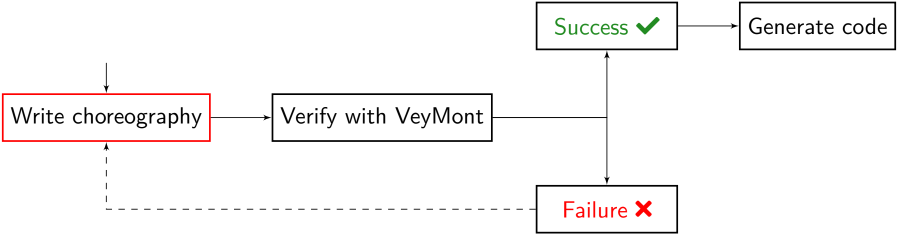
- Only local memory is supported
  - No ownership transfer
  - No read sharing
- Annotations are not preserved in generated implementation
  - Cannot verify generated implementation
  - Cannot analyze generated implementation (e.g. Alpinist [4])
  - More safely extend beyond metatheory

Contributions

- Extend VeyMont with **endpoint ownership annotations**
  - This enables fine-grained memory specification
- A transform into regular permissions called **stratified permissions**
- **Extend endpoint projection** to be annotation-aware
- Illustrate with **Tic-Tac-Toe case study**

Endpoint ownership annotations

Syntax


Success ✔
Generate code
Write choreography
Verify with VeyMont
Failure ✘

Basic choreographies

```
choreography incr(Store s) {
  endpoint a = Role(s);
  endpoint b = Role(null);



  run {
    a.s.x := a.s.x + 1;


    communicate a.s -> b.s;
    b.s.x := b.s.x + 1;


    communicate b.s -> a.s;
  }
}
```

Functional correctness annotations

```
choreography incr(Store s) {
  endpoint a = Role(s);
  endpoint b = Role(null);

  requires a.s.x > 0;
  ensures a.s.x > 2;
  run {
    a.s.x := a.s.x + 1;


    communicate a.s -> b.s;
    b.s.x := b.s.x + 1;


    communicate b.s -> a.s;
  }
}
```

Aside: permissions

Indicate write permission for a heap location:

```
Perm(o.f, 1)
```

Positive fraction is read permission:

```
Perm(o.f, 1\2)
```

Compose using `**` operator:

```
Perm(o.f, 1) ** Perm(x.y, 1\2);
```

Split and join:

`Perm(o.f, 1)` $\equiv$ `Perm(o.f, 1\2) ** Perm(o.f, 1\2)`

Aside: permissions

Indicate write permission for a heap location:

```
Perm(o.f, 1)
```

Positive fraction is read permission:

```
Perm(o.f, 1\2)
```

Compose using ** operator:

```
Perm(o.f, 1) ** Perm(x.y, 1\2);
```

Split and join:

```
Perm(o.f, 1) ≡ Perm(o.f, 1\2) ** Perm(o.f, 1\2)
```

Extension:

```
Perm[e](o.f, 1)
```

Permission is owned by e

```
(\endpoint e; e.x > 0)
```

Expression is only evaluated to e

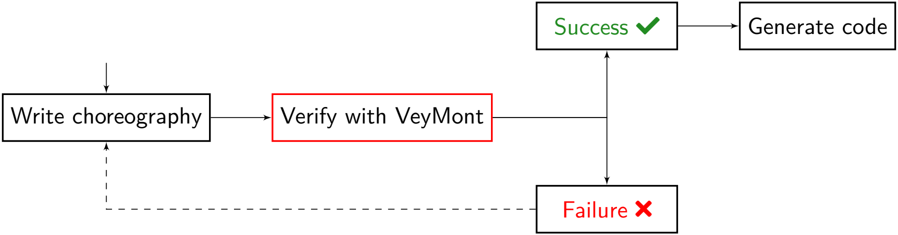
Endpoint ownership

```
choreography incr(Store s) {
  endpoint a = Role(s);
  endpoint b = Role(null);

  requires Perm[a](a.s.x, 1) ** (\endpoint a; a.s.x > 0)
  ensures Perm[a](a.s.x, 1) ** (\endpoint a; a.s.x > 2);
  run {
    a.s.x := a.s.x + 1;
    channel_invariant
      Perm(\msg.x, 1) ** \msg.x > 1;
    communicate a.s -> b.s;
    b.s.x := b.s.x + 1;
    channel_invariant
      Perm(\msg.x, 1) ** \msg.x > 2;
    communicate b.s -> a.s;
  }
}
```

Stratified permissions

Verification



Stratified permissions

- Q: How to verify endpoint ownership annotations?

Stratified permissions

- Q: How to verify endpoint ownership annotations?
- A: By transforming into regular ownership annotations.

Stratified permissions

- Q: How to verify endpoint ownership annotations?
- A: By transforming into regular ownership annotations.

Wrap each permission in a resource predicate that has an extra argument for the endpoint owner.

Aside: resource predicates

- Special type of permission that contains other permissions
- Essence: hide permissions and constraints behind opaque name
- Exchange name $\iff$ permissions and constraints using `fold`/`unfold`

Resource predicates example: `unfold`

```
resource positive(Store o) =
  Perm(o.f, 1) ** o.f > 0;

requires positive(o);
void m(Store o) {
  // Next assert fails:
  assert Perm(o.f, 1) ** o.f > 0;
}
```

Resource predicates example: `unfold`

```
resource positive(Store o) =
  Perm(o.f, 1) ** o.f > 0;

requires positive(o);
void m(Store o) {
  unfold positive(o);
  assert Perm(o.f, 1) ** o.f > 0;
}
```

Resource predicates example: fold

```
resource positive(Store o) =
  Perm(o.f, 1) ** o.f > 0;

requires Perm(o.f, 1) ** o.f > 0;
void m(Store o) {
  // Next assert fails:
  assert positive(o);
}
```

Resource predicates example: fold

```
resource positive(Store o) =
  Perm(o.f, 1) ** o.f > 0;

requires Perm(o.f, 1) ** o.f > 0;
void m(Store o) {
  fold positive(o);
  assert positive(o);
}
```

Using predicates for stratified permissions

For each field `.f` of class `C`, generate:

```
resource wrapF(any endpointOwner, C c) =
  Perm(c.f, 1);
```

For each `Perm[alex](c.f, 1)`, replace with:

```
wrapF(alex, c)
```

Interacting with stratified permissions (bonus!)

Wrapped stratified permissions are only unfolded in two situations:

1. When endpoint does local computation
2. When endpoints exchange message + permissions

Interacting with stratified permissions (bonus!)

Wrapped stratified permissions are only unfolded in two situations:

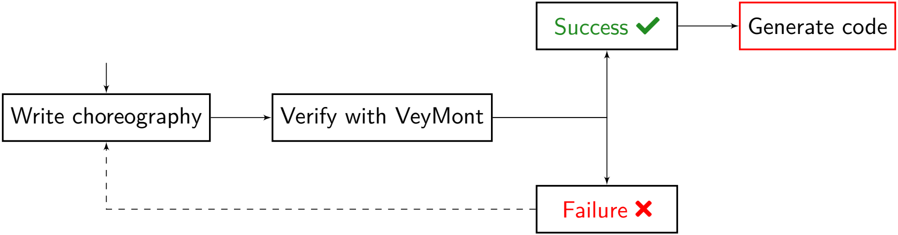
1. When endpoint does local computation
2. When endpoints exchange message + permissions

```
alex.x := bob.y + 1;
```

```
// Compute:
int v =
  (unfolding
     wrapY(alex, bob)
   in bob.y) + 1;
// Store:
unfold wrapX(alex, alex);
alex.x = v;
fold wrapX(alex, alex);
```

Extending endpoint projection

Verification



Original endpoint projection

Original rules for endpoint a:

`a.x := 5;`	$\rightarrow$	`a.x = 5;`
`communicate a.x -> b.y;`	$\rightarrow$	`a_b.writeValue(a.x);`
`if (a.x == 5 && b.y == 9)`	$\rightarrow$	`if (a.x == 5 && true)`
`a.foo(a.x)`	$\rightarrow$	`a.foo(a.x)`
`b.bar(b.x)`	$\rightarrow$	`/* skip */`

Endpoint projection supporting annotations

(For endpoint a:)

For statements:

```
b: a.x := 5;                      →        /* skip */
communicate b: a.x -> a: b.y;     →   b.y = a_b.readValue
```

For expressions:

```
Perm[a](x.f, 1)        →   Perm(x.f, 1)
Perm[b](x.f, 1)        →   true
(\endpoint a; expr)    →   expr
(\endpoint b; expr)    →   true
```

Tic-Tac-Toe case studies

Case studies

Name	Verifies	Impl verifies	Remark
TTT	✓	✗	Baseline
$\mathrm{TTT}_{\mathrm{msg}}$	✓	✓	One message runtime overhead
$\mathrm{TTT}_{\mathrm{last}}$	✓	✓	No runtime overhead

Case study 1: TTT

```
while(!p1.gameFinished() && !p2.gameFinished()) {
  if(p1.turn && !p2.turn) {
    p1.createNewMove();
    p1.doMove();
    channel_invariant ...;
    communicate p2.move <- p1.move.copy();

  } else {
    p2.createNewMove();
    p2.doMove();
    channel_invariant ...;
    communicate p1.move <- p2.move.copy();
    p1.doMove();
  }
  p1.turn := !p1.turn;
  p2.turn := !p2.turn;
}
```

Case study 2: TTT_{msg}

```
inline resource turnInvariant(Player p, Player q) =
  ([1\2]p.boardPerm()) **
  ([1\2]q.boardPerm()) **
  p.equalBoard(q);

// Every turn:
channel_invariant
  \msg.state() **
  ([1\2]\sender.boardPerm()) **
  ([1\2]\receiver.boardPerm()) **
  \sender.oneMoveAheadOf(\msg, \receiver);
communicate p2.move <- p1.move.copy();

// Game finished:
channel_invariant
  [1\4]\sender.boardPerm() **
  [1\4]\receiver.boardPerm() **
  \receiver.equalBoard(\sender);
communicate p2.temp <- p1: true;
```

Case study 3: $\mathrm{TTT}_{\mathrm{last}}$

```
// After making a move:
if (p1.gameFinished()) {
  p1.returner.mark := 1 - p1.myMark;
}
channel_invariant
  ... **
  (!\sender.gameFinished() ==>
    ([1\2]\sender.returnerMark()));
communicate p2.move <- p1.move.copy();

// Postcondition:
ensures (\endpoint p1;
  ([1\2]p1.returnerMark()) **
  (p1.returner.mark == p1.myMark ==>
    /* permissions and boards are equal */));
```

Conclusion

Conclusion

- VeyMont: correctness by construction, verification as a recurring component, fine-grained memory management
- How: using this one weird trick: endpoint ownership annotations
- Transform endpoint annotations into regular permissions using **stratified permissions**
- **Extended endpoint projection** to be annotation-aware
- **Tic-Tac-Toe case study** illustrated usability

Robert Rubbens
Formal Methods & Tools, University of Twente
r.b.rubbens@utwente.nl

Bonus slides

Bibliography I

[1] Stefan Blom et al. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". 2017. DOI: 10.1007/978-3-319-66845-1_7.

[2] Petra van den Bos and Sung-Shik Jongmans. "VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs". 2023. DOI: 10.1007/978-3-031-27481-7_19.

[3] Sung-Shik Jongmans and Petra van den Bos. "A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming". 2022. DOI: 10.1007/978-3-030-99336-8_19.

[4] Ömer Sakar et al. "Alpinist: An Annotation-Aware GPU Program Optimizer". 2022. DOI: 10.1007/978-3-030-99527-0_18.

VerCors & VeyMont

Soundness?

- In theory: no, e.g.:

```
channel_invariant
  Perm(a.ghostX, 1\2) **
  a.ghostX == \msg;
communicate a.x -> b.x;
// ... omitted computation from a
assert (\endpoint a; isPrime(a.ghostX)); // #1
// Can b now also conclude?
assert (\endpoint b; isPrime(b.x));      // #2
```

- In practice: kind of

Case study 1: TTT

```
loop_invariant (\chor ticTacToeAnnotations(p1, p2));
loop_invariant p1.myMark == 0 ** p2.myMark == 1;
while(!p1.gameFinished() && !p2.gameFinished()) {
  if(p1.turn && !p2.turn) {
    p1.createNewMove();
    p1.doMove();
    channel_invariant
      (\chor \msg.i == \sender.move.i &&
        \msg.j == \sender.move.j &&
        \msg.mark == \sender.move.mark);
    communicate p2.move <- p1.move.copy();

  } else {
    p2.createNewMove();
    p2.doMove();
    channel_invariant
      (\chor \msg.i == \sender.move.i &&
        \msg.j == \sender.move.j &&
        \msg.mark == \sender.move.mark);
    communicate p1.move <- p2.move.copy();
    p1.doMove();
  }
  p1.turn := !p1.turn;
  p2.turn := !p2.turn;
}
```

Local computation example

```
alex.x := bob.y + 1;
```

$$\Downarrow$$

```
resource wrapX(any endpoint_owner, Store s) =
  Perm(s.x, 1);
resource wrapY(any endpoint_owner, Store s) =
  Perm(s.y, 1);

// Compute:
int v =
  (unfolding
     wrapY(alex, bob)
   in bob.y) + 1;
// Store:
unfold wrapX(alex, alex);
alex.x = v;
fold wrapX(alex, alex);
```

Message & permission exchange example

```
channel_invariant Perm(s.x, 1) ** s.x > 0;
communicate alex.s -> bob.s;
```

$$\Downarrow$$

```
resource wrapX(any endpoint_owner, Store s) = Perm(s.x, 1);
resource wrapS(any endpoint_owner, Role r) = Perm(r.s, 1);

// Compute message
int m = (unfolding wrapS(alex, alex) in alex.s)

// Check & remove invariant from alex's state
exhale wrapX(alex, m) ** (unfolding wrapX(alex, m) in m.x) > 0;

// Add invariant to bob's state
inhale wrapX(bob, m) ** (unfolding wrapX(bob, m) in m.x) > 0;

// Store message in bob's field
unfold wrapS(bob, bob);
bob.s = m;
fold wrapS(bob, bob);
```