

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

Robert¹ Rubbens, Petra van den Bos, Marieke Huisman

FMT Colloquium, Enschede

November 15th, 2024

UNIVERSITY
OF TWENTE.



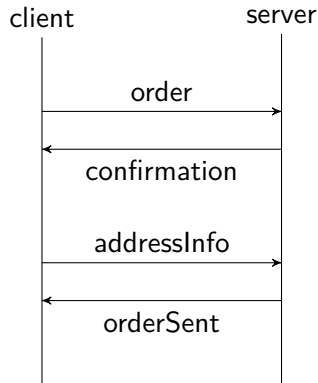
¹(Bob)



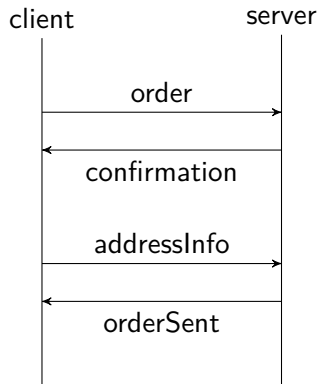
- Starting point: VeyMont: correctness by construction, single-owner memory management, verification
- Our contribution: verification as a recurring component, fine-grained memory
- How: using this one weird trick: endpoint ownership annotations
 - ~ indicating per expression the relevant endpoint



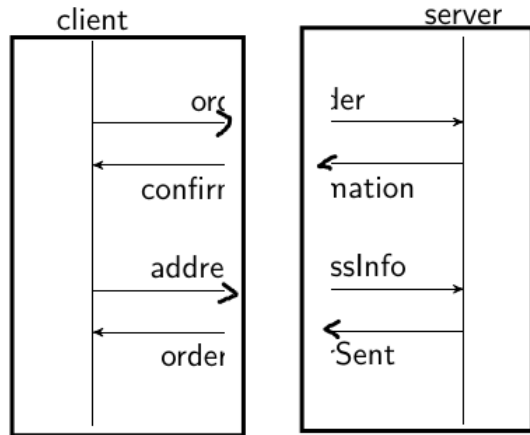
- Systems grow larger every day
 - More computers
 - More threads
- One solution: choreographies
- Choreography: series of message exchanges between 2+ parties
- Benefits:
 - Message fidelity
 - Deadlock freedom
 - Code generation



Choreographies: code generation



$\Rightarrow$
endpoint
projection



Choreographies: notation



```
1 choreography incr(int x) {
2   endpoint alex = Role(x);
3   endpoint bob  = Role(-1);
4
5   run {
6     alex.v := alex.v + 1;
7     communicate alex.v -> bob.v;
8     bob.v := bob.v + 1;
9     communicate bob.v -> alex.v;
10  }
11 }
```

Choreographies: notation



```
1 choreography incr(int x) {  
2   endpoint alex = Role(x);  
3   endpoint bob  = Role(-1);  
4  
5   run {  
6     alex.v := alex.v + 1;  
7     communicate alex.v -> bob.v;  
8     bob.v := bob.v + 1;  
9     communicate bob.v -> alex.v;  
10  }  
11 }
```

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



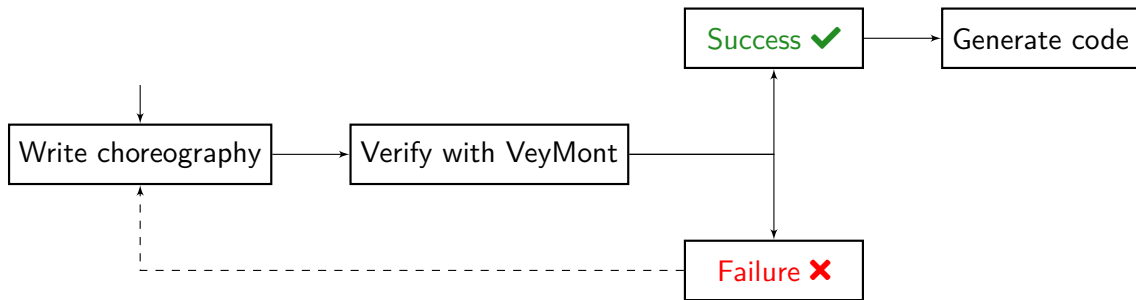
- Beyond deadlock freedom & message fidelity
- Assertions, pre- and postconditions

Metatheory [3]:

Choreography correct $\implies$ generated implementation correct



- 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



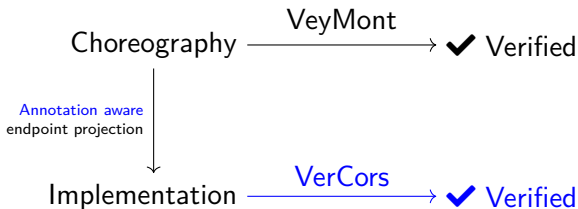
```
1 choreography incr(int x) {
2   endpoint alex = Role(x);
3   endpoint bob  = Role(-1);
4
5   requires alex.v > 0;
6   ensures alex.v > 2;
7   run {
8     alex.v := alex.v + 1;
9     communicate alex.v -> bob.v;
10    bob.v := bob.v + 1;
11    communicate bob.v -> alex.v;
12  }
13 }
```



- 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

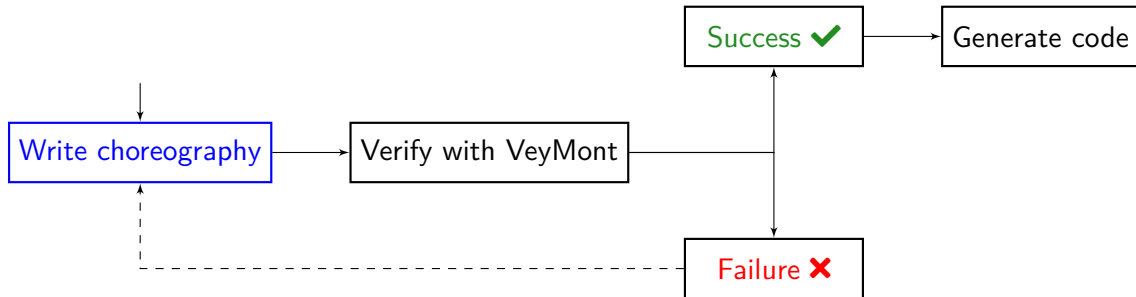


- Extend VeyMont with **endpoint ownership annotations**
 - This enables fine-grained memory specification
- A transform into regular permissions called **stratified permissions**
- **Extend endpoint projection** to be annotation-aware
- Illustrate with **Tic-Tac-Toe case study**



Endpoint ownership annotations





Basic choreographies



```
1 choreography incr(Store s) {
2   endpoint a = Role(s);
3   endpoint b = Role(null);
4
5
6
7   run {
8     a.s.v := a.s.v + 1;
9
10
11     communicate a.s -> b.s;
12     b.s.v := b.s.v + 1;
13
14
15     communicate b.s -> a.s;
16   }
17 }
```

Functional correctness annotations



```
1 choreography incr(Store s) {
2   endpoint a = Role(s);
3   endpoint b = Role(null);
4
5   requires a.s.v > 0;
6   ensures a.s.v > 2;
7   run {
8     a.s.v := a.s.v + 1;
9
10
11     communicate a.s -> b.s;
12     b.s.v := b.s.v + 1;
13
14
15     communicate b.s -> a.s;
16   }
17 }
```

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



```
1 choreography incr(Store s) {
2   endpoint a = Role(s);
3   endpoint b = Role(null);
4
5   requires Perm[a](a.s.v, 1) ** (\endpoint a; a.s.v > 0);
6   ensures Perm[a](a.s.v, 1) ** (\endpoint a; a.s.v > 2);
7   run {
8     a.s.v := a.s.v + 1;
```

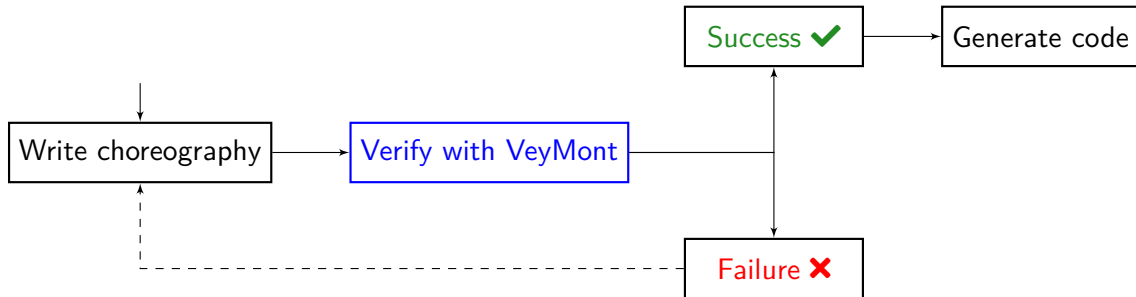
Endpoint ownership



```
1 choreography incr(Store s) {
2   endpoint a = Role(s);
3   endpoint b = Role(null);
4
5   requires Perm[a](a.s.v, 1) ** (\endpoint a; a.s.v > 0);
6   ensures Perm[a](a.s.v, 1) ** (\endpoint a; a.s.v > 2);
7   run {
8     a.s.v := a.s.v + 1;
9     channel_invariant
10      Perm(\msg.v, 1) ** \msg.v > 1;
11    communicate a.s -> b.s;
12    b.s.v := b.s.v + 1;
13    channel_invariant
14      Perm(\msg.v, 1) ** \msg.v > 2;
15    communicate b.s -> a.s;
16  }
17 }
```

Stratified permissions







- Q: How to verify endpoint ownership annotations?



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



- 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.



- 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



```
1 resource positive(Store o) =  
2   Perm(o.f, 1) ** o.f > 0;  
3  
4 requires positive(o);  
5 void m(Store o) {  
6   // Next assert fails:  
7   assert Perm(o.f, 1) ** o.f > 0;  
8 }
```

Resource predicates example: unfold



```
1 resource positive(Store o) =  
2   Perm(o.f, 1) ** o.f > 0;  
3  
4 requires positive(o);  
5 void m(Store o) {  
6   unfold positive(o);  
7   assert Perm(o.f, 1) ** o.f > 0;  
8 }
```

Resource predicates example: fold



```
1 resource positive(Store o) =  
2   Perm(o.f, 1) ** o.f > 0;  
3  
4 requires Perm(o.f, 1) ** o.f > 0;  
5 void m(Store o) {  
6   // Next assert fails:  
7   assert positive(o);  
8 }
```

Resource predicates example: fold



```
1 resource positive(Store o) =  
2   Perm(o.f, 1) ** o.f > 0;  
3  
4 requires Perm(o.f, 1) ** o.f > 0;  
5 void m(Store o) {  
6   fold positive(o);  
7   assert positive(o);  
8 }
```

Using predicates for stratified permissions



E.g. for some permission with an endpoint owner:

```
1 requires Perm[alex](c.f, 1);
```

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

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

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

```
4 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!)



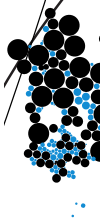
Wrapped stratified permissions are only unfolded in two situations:

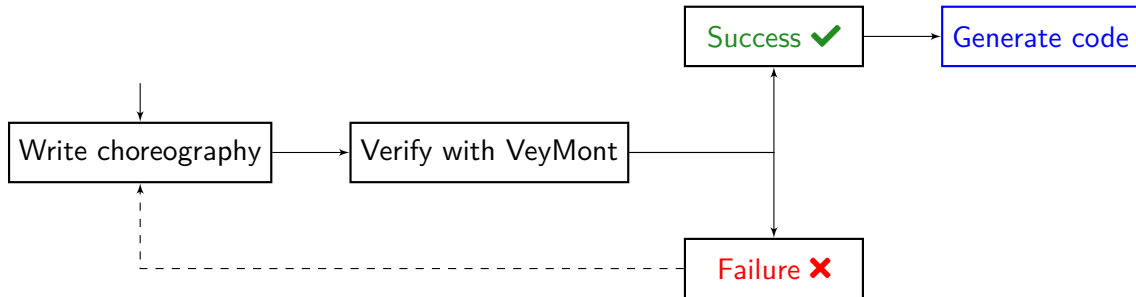
- 1 When endpoint does local computation
- 2 When endpoints exchange message + permissions

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

1  // Compute:
2  int v =
3      (unfolding
4          wrapY(alex, bob)
5          in bob.y) + 1;
6  // Store:
7  unfold wrapZ(alex, alex);
8  alex.z = v;
9  fold wrapZ(alex, alex);
```

Extending endpoint projection





Original endpoint projection



Original rules for endpoint a:

<code>a.x := 5;</code>	<code>→</code>	<code>a.x = 5;</code>
<code>communicate a.x -> b.y;</code>	<code>→</code>	<code>a_b.writeValue(a.x);</code>
<code>if (a.x == 5 && b.y == 9)</code>	<code>→</code>	<code>if (a.x == 5 && true)</code>
<code> a.foo(a.x)</code>	<code>→</code>	<code> a.foo(a.x)</code>
<code> b.bar(b.x)</code>	<code>→</code>	<code> /* skip */</code>

Endpoint projection supporting annotations



(For endpoint a:)

For statements:

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





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

Case study 1: TTT



```
1  while(!p1.gameFinished() && !p2.gameFinished()) {
2    if(p1.turn && !p2.turn) {
3      p1.createNewMove();
4      p1.doMove();
5      channel_invariant ...;
6      communicate p2.move <- p1.move.copy();
7      p2.doMove();
8    } else {
9      p2.createNewMove();
10     p2.doMove();
11     channel_invariant ...;
12     communicate p1.move <- p2.move.copy();
13     p1.doMove();
14   }
15   p1.turn := !p1.turn;
16   p2.turn := !p2.turn;
17 }
```

Case study 2: TTT_{msg}



```
1 inline resource turnInvariant(Player p, Player q) =
2   ([1\2]p.boardPerm()) **
3   ([1\2]q.boardPerm()) **
4   p.equalBoard(q);
5
6 // Every turn:
7 channel_invariant
8   \msg.state() **
9   ([1\2]\sender.boardPerm()) **
10  ([1\2]\receiver.boardPerm()) **
11  \sender.oneMoveAheadOf(\msg, \receiver);
12 communicate p2.move <- p1.move.copy();
13
14 // Game finished:
15 channel_invariant
16   [1\4]\sender.boardPerm() **
17   [1\4]\receiver.boardPerm() **
18   \receiver.equalBoard(\sender);
19 communicate p2.temp <- p1: true;
```

Case study 3: TTT_{last}



```
1 // After making a move:
2 if (p1.gameFinished()) {
3     p1.returner.mark := 1 - p1.myMark;
4 }
5
6 // Postcondition:
7 ensures (\endpoint p1;
8     ([1\2]p1.returnerMark()) **
9     (p1.returner.mark == p1.myMark ==>
10         /* 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

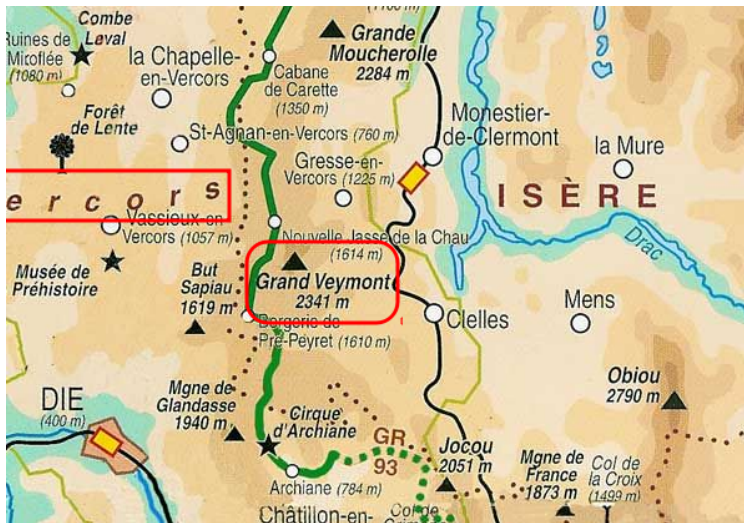


Bonus slides



- [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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1007/978-3-030-99527-0_18).

VerCors & VeyMont





- In theory: no, e.g.:

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

- In practice: kind of

Case study 1: TTT



```
1 loop_invariant (\chor ticTacToeAnnotations(p1, p2));
2 loop_invariant p1.myMark == 0 ** p2.myMark == 1;
3 while(!p1.gameFinished() && !p2.gameFinished()) {
4   if(p1.turn && !p2.turn) {
5     p1.createNewMove();
6     p1.doMove();
7     channel_invariant
8       (\chor \msg.i == \sender.move.i &&
9         \msg.j == \sender.move.j &&
10        \msg.mark == \sender.move.mark);
11    communicate p2.move <- p1.move.copy();
12    p2.doMove();
13  } else {
14    p2.createNewMove();
15    p2.doMove();
16    channel_invariant
17      (\chor \msg.i == \sender.move.i &&
18        \msg.j == \sender.move.j &&
19        \msg.mark == \sender.move.mark);
20    communicate p1.move <- p2.move.copy();
21    p1.doMove();
22  }
23  p1.turn := !p1.turn;
24  p2.turn := !p2.turn;
25 }
```

Local computation example



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



```
1 resource wrapX(any endpoint_owner, Store s) =  
2   Perm(s.x, 1);  
3 resource wrapY(any endpoint_owner, Store s) =  
4   Perm(s.y, 1);  
5  
6 // Compute:  
7 int v =  
8   (unfolding  
9     wrapY(alex, bob)  
10    in bob.y) + 1;  
11 // Store:  
12 unfold wrapX(alex, alex);  
13 alex.x = v;  
14 fold wrapX(alex, alex);
```

Message & permission exchange example



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

      ↓↓

1 resource wrapX(any endpoint_owner, Store s) = Perm(s.x, 1);
2 resource wrapS(any endpoint_owner, Role r) = Perm(r.s, 1);
3
4 // Compute message
5 int m = (unfolding wrapS(alex, alex) in alex.s)
6
7 // Check & remove invariant from alex's state
8 exhale wrapX(alex, m) ** (unfolding wrapX(alex, m) in m.x) > 0;
9
10 // Add invariant to bob's state
11 inhale wrapX(bob, m) ** (unfolding wrapX(bob, m) in m.x) > 0;
12
13 // Store message in bob's field
14 unfold wrapS(bob, bob);
15 bob.s = m;
16 fold wrapS(bob, bob);
```