VeyMont: Verification of Choreographic Programs with Permissions and Parameterization

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

February 6th, 2024

[1](Bob)

Takeaways:

1. VeyMont: why & how?
   - Parallelises verified programs instead of verifying parallel programs
   - Verify choreographies (think of: session types)
   - Generate executable code
2. Now: permission support
   - VeyMont generates permissions annotations
   - Shortens already short programs.
   - But: limits the programs that verify
3. Future work: parameterization
   - Key insight: parallel blocks
   - Case study on distributed elect/summation done
   - No tool support yet

Takeaways:

1. VeyMont: why & how?
   - Parallelises verified programs instead of verifying parallel programs
   - Verify choreographies (think of: session types)
   - Generate executable code
2. Now: permission support
   - VeyMont generates permissions annotations
   - Shortens already short programs.
   - But: limits the programs that verify
3. Future work: parameterization
   - Key insight: parallel blocks
   - Case study on distributed elect/summation done
   - No tool support yet

Takeaways:

1. VeyMont: why & how?
   - Parallelises verified programs instead of verifying parallel programs
   - Verify choreographies (think of: session types)
   - Generate executable code
2. Now: permission support
   - VeyMont generates permissions annotations
   - Shortens already short programs.
   - But: limits the programs that verify
3. Future work: parameterization
   - Key insight: parallel blocks
   - Case study on distributed elect/summation done
   - No tool support yet

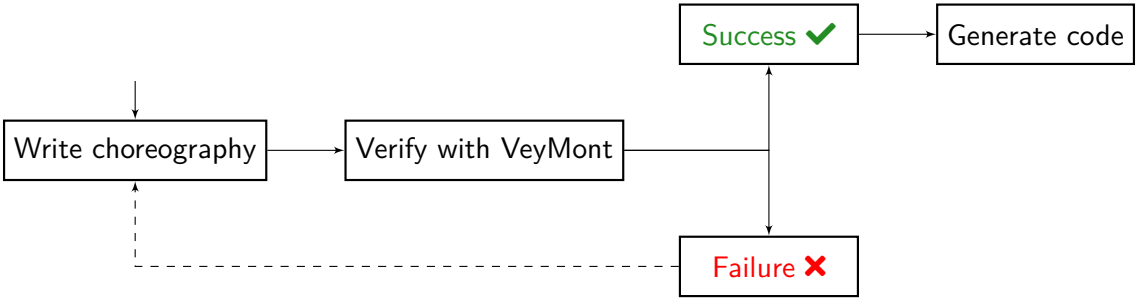
VeyMont: why & how?

VeyMont

- Verifier built on top of VerCors [1]
- Verifies your choreography, then generates code
- Nice: looks kind-of sequential, sometimes easier
- “Parallelise verified programs”

VeyMont

- Verifier built on top of VerCors [1]
- Verifies your choreography, then generates code
- Nice: looks kind-of sequential, sometimes easier
- “Parallelise verified programs”



swap in VeyMont

```
1  seq_program swap(int x, int y) {
2    endpoint alex = Role(x);
3    endpoint bob  = Role(y);
4
5
6
7    seq_run {
8      communicate alex.temp <- bob.v;
9      communicate bob.temp <- alex.v;
10     alex.v := alex.temp;
11     bob.v := bob.temp;
12   }
13 }
```

swap in VeyMont

```
1  seq_program swap(int x, int y) {
2    endpoint alex = Role(x);
3    endpoint bob  = Role(y);
4
5
6
7    seq_run {
8      communicate alex.temp <- bob.v;
9      communicate bob.temp <- alex.v;
10     alex.v := alex.temp;
11     bob.v := bob.temp;
12   }
13 }
```

Memory safety, deadlock freedom

swap in VeyMont, verified

```
1  seq_program swap(int x, int y) {
2    endpoint alex = Role(x);
3    endpoint bob  = Role(y);
4
5    ensures alex.v == \old(bob.v);
6    ensures bob.v == \old(alex.v);
7    seq_run {
8      communicate alex.temp <- bob.v;
9      communicate bob.temp <- alex.v;
10     alex.v := alex.temp;
11     bob.v := bob.temp;
12   }
13 }
```

swap in VeyMont, verified

```
1  seq_program swap(int x, int y) {
2    endpoint alex = Role(x);
3    endpoint bob  = Role(y);
4
5    ensures alex.v == \old(bob.v);
6    ensures bob.v == \old(alex.v);
7    seq_run {
8      communicate alex.temp <- bob.v;
9      communicate bob.temp <- alex.v;
10     alex.v := alex.temp;
11     bob.v := bob.temp;
12   }
13 }
```

Deadlock freedom, memory safety, functional correctness

swap in VeyMont, generated

Listing 1: Implementation for Alex

```
1  void alex_run(
2      Role a,
3      Role b,
4      Chan  a_b,
5      Chan b_a) {
6    a.temp = b_a.read();
7    a_b.write(a.v);
8    a.v = a.temp;
9    // Skip write for Bob
10 }
```

Listing 2: Implementation for Bob

```
1  void bob_run(
2      Role a,
3      Role b,
4      Chan a_b,
5      Chan b_a) {
6    b_a.write(b.v);
7    b.temp = a_b.read();
8    // Skip write for alex
9    b.v = b.temp;
10 }
```

swap in VeyMont, generated

Listing 1: Implementation for Alex

```
1  void alex_run(
2      Role a,
3      Role b,
4      Chan  a_b,
5      Chan b_a) {
6    a.temp = b_a.read();
7    a_b.write(a.v);
8    a.v = a.temp;
9    // Skip write for Bob
10 }
```

Listing 2: Implementation for Bob

```
1  void bob_run(
2      Role a,
3      Role b,
4      Chan a_b,
5      Chan b_a) {
6    b_a.write(b.v);
7    b.temp = a_b.read();
8    // Skip write for alex
9    b.v = b.temp;
10 }
```

- Deadlock freedom, functional correctness via metatheory
- Executable

Aside: permissions?

- Where do the generated permissions go?
- VeyMont: built on top of VerCors [1]
- Verifier for concurrent programs in Java, C, OpenCL/CUDA
- Typically we use PVL
- Frontends: Alpinist (optimizations) [4], VeSUV (embedded) [5]
- Uses permission-based separation logic

VerCors & permissions in a nutshell

```
1
2 void bob_run(Role a, Role b,
3     Chan a_b, Chan b_a) {
4   b_a.write(bob.v);
5
6   b.temp = a_b.read();
7   b.v = b.temp;
8 }
```

VerCors & permissions in a nutshell

```
1
2 void bob_run(Role a, Role b,
3     Chan a_b, Chan b_a) {
4   b_a.write(bob.v);
5   // Error: no permission for b.temp
6   b.temp = a_b.read();
7   b.v = b.temp;
8 }
```

VerCors & permissions in a nutshell

```
1 requires true;
2 void bob_run(Role a, Role b,
3     Chan a_b, Chan b_a) {
4   b_a.write(bob.v);
5   // Error: no permission for b.temp
6   b.temp = a_b.read();
7   b.v = b.temp;
8 }
```

VerCors & permissions in a nutshell

```
1 requires Perm(b.temp, 1) ** Perm(b.v, 1);
2 void bob_run(Role a, Role b,
3     Chan a_b, Chan b_a) {
4   b_a.write(bob.v);
5
6   b.temp = a_b.read();
7   b.v = b.temp;
8 }
```

VerCors & permissions in a nutshell

```
1 requires Perm(b.temp, 1) ** Perm(b.v, 1);
2 void bob_run(Role a, Role b,
3     Chan a_b, Chan b_a) {
4   b_a.write(bob.v);
5   // All good
6   b.temp = a_b.read();
7   b.v = b.temp;
8 }
```

Other contract annotations

- `requires P; ensures Q;`
- `loop_invariant I; while (b) { c; }`
- `resource p(Role r) = Perm(r.temp, 1) ** r.temp == 0;`
  - Package with `fold`, open up with `unfold`
- `lock_invariant` for locks

Work so far

- "VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs" (Petra van den Bos and Sung-Shik Jongmans, 2023) [2]
- "A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming" (Sung-Shik Jongmans and Petra van den Bos, 2022) [3]

VeyMont: room for improvement

- Minor aspects:
  - Cannot verify generated implementation
  - Would like to make changes afterwards, reverify
  - Analyse output in other tools, e.g. perhaps Alpinist?
  - Enhance supported language while not redoing metatheory

VeyMont: room for improvement

- Minor aspects:
  - Cannot verify generated implementation
  - Would like to make changes afterwards, reverify
  - Analyse output in other tools, e.g. perhaps Alpinist?
  - Enhance supported language while not redoing metatheory
- Major aspects:
  1. Permission support for fine grained memory sharing
  2. Parameterization

Permission support

Why extra permission support?

Previous example worked fine without permissions?

Why extra permission support?

Previous example worked fine without permissions?

Current limitations:

- Message passing for information sharing $\Rightarrow$ runtime overhead
- Incorporate permissions from other parts of the program
- Generation is incomplete

Why extra permission support?

Previous example worked fine without permissions?

Current limitations:

- Message passing for information sharing $\Rightarrow$ runtime overhead
- Incorporate permissions from other parts of the program
- Generation is incomplete

In addition: verification of generated program

Solution

More annotations!

- For verification:
  - Endpoint owner of permission
  - Channel invariant to specify transfer of ownership
- For generation:
  - Annotate expression with endpoint context

Example: shared memory

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm(s.x, 1);
7    seq_run {
8      s.x := 5;  // Who has permission?
9      s.x := 10; // Who is assigning?
10   }
11 }
```

Add: permission owner

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm[alex](s.x, 1);
7    seq_run {
8      s.x := 5;
9      s.x := 10; // Who is assigning?
10   }
11 }
```

Add: assignment context

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm[alex](s.x, 1);
7    seq_run {
8      alex: s.x := 5;  // Ok: alex has permission
9      bob:  s.x := 10; // Error: no permission
10   }
11 }
```

Channel invariant

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm[alex](s.x, 1);
7    ensures Perm[bob](s.x, 1);
8    seq_run {
9      alex: s.x := 5;
10     channel_invariant Perm(s.x, 1);
11     communicate bob.v <- alex.v; // Transfer
12     bob:  s.x := 10; // Ok: bob has permission
13   }
14 }
```

Add: functional spec

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm[alex](s.x, 1);
7    ensures Perm[bob](s.x, 1);
8    ensures s.x == 10; // Who is evaluating?
9    seq_run {
10     alex: s.x := 5;
11     channel_invariant Perm(s.x, 1);
12     communicate bob.v <- alex.v;
13     bob:  s.x := 10;
14   }
15 }
```

Add: expression context

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm[alex](s.x, 1);
7    ensures Perm[bob](s.x, 1);
8    ensures (\in_endpoint bob; s.x == 10);
9    seq_run {
10     alex: s.x := 5;
11     channel_invariant Perm(s.x, 1);
12     communicate bob.v <- alex.v;
13     bob:  s.x := 10;
14   }
15 }
```

Implementation challenge

```
1  requires Perm(s.x, 1);
2  seq_program Example(Store s) {
3    endpoint alex = Role();
4    endpoint bob = Role();
5
6    requires Perm[alex](s.x, 1);
7    seq_run {
8      bob: s.x := 5; // Expect error
9    }
10 }
```

Incomplete encoding

```
1 requires Perm[alex](s.x, 1);
2 seq_run {
3   bob: s.x := 5; // Expect error
4 }
```

Incomplete encoding

```
1 requires Perm[alex](s.x, 1);
2 seq_run {
3   bob: s.x := 5; // Expect error
4 }
```

```
1 requires Perm(s.x, 1);
2 void run_encoded(Role alex, Role bob, Store s) {
3   s.x = 5; // Verifies: wrong
4 }
```

Incomplete encoding

```
1 requires Perm[alex](s.x, 1);
2 seq_run {
3   bob: s.x := 5; // Expect error
4 }
```

```
1 requires Perm(s.x, 1);
2 void run_encoded(Role alex, Role bob, Store s) {
3   s.x = 5; // Verifies: wrong
4 }
```

Cannot just discard owner when encoding!

Encoding permission owner

```
1 requires Perm[alex](s.x, 1);
2 seq_run {
3   bob: s.x := 5; // Error: no permission
4 }
```

⇓

```
1 resource wrapped(Role owner, Store s) =
2   Perm(s.x, 1);
3
4 requires wrapped(alex, s);
5 void run_encoded(Role alex, Role bob, Store s) {
6   unfold wrapped(bob, s); // Error
7   s.x = 5;
8   fold wrapped(bob, s);
9 }
```

Parameterization

Ongoing work: parameterization

- Many useful algorithms: leader election, distributed summation.
- Useful primitive in VerCors: the `par` block
- How do we integrate this insight into VeyMont?

VerCors: par block

```
1 par(int tid = 0 .. N)
2   requires Perm(a[tid], write);
3   requires Perm(b[tid], read);
4   requires Perm(c[tid], read);
5   ensures a[tid] == b[tid] + c[tid];
6 {
7   a[tid] = b[tid]+c[tid];
8 }
```

VeyMont: par block...?

```
1 par (int i = 0 .. N - 1 )
2   requires ...;
3 {
4   communicate nodes[i].x <- nodes[i + 1].x;
5 }
```

VeyMont: par block...?

```
1 par (int i = 0 .. N - 1 )
2   requires ...;
3 {
4   communicate nodes[i].x <- nodes[i + 1].x;
5 }
```

Already enough for e.g. ring algorithms:

```
1 par (int i = 0 .. N - 1 )
2   requires ...;  // Hmm...
3 {
4   communicate nodes[i + 1].x <- nodes[i].x;
5 }
6
7 communicate nodes[0].x <- nodes[N - 1].x;
```

Parameterizing VeyMont

What is necessary?

- Parameterizing `endpoint` declarations
- Parameterizing `communicate` statements

Parameterized syntax summary

```
1 seq_program Example(Store s, int N) {
2   endpoint[tid: 0 .. N] roles = Role(tid);
3
4   seq_run {
5     communicate
6       role[i: 0 .. N - 1].v <- role[i + 1].v;
7     roles[i: 0 .. N]: s.xs[i] := 10;
8   }
9 }
```

Parameterized syntax summary

```
1 seq_program Example(Store s, int N) {
2   endpoint[tid: 0 .. N] roles = Role(tid);
3
4   seq_run {
5     communicate
6       role[i: 0 .. N - 1].v <- role[i + 1].v;
7     roles[i: 0 .. N]: s.xs[i] := 10;
8   }
9 }
```

How to verify? How to generate code?

Verifying parameterized choreographies (1)

Endpoint family translation:

```
1 endpoint[tid: 0 .. N] roles = Role(tid);
2 ⇓
3 seq<Role> roles = /* empty seq */;
4 for(int i = 0 .. N) { ... };
```

Verifying parameterized choreographies (2)

Parameterized communicate:

```
1  channel_invariant I(\sender, \receiver, \msg);
2  communicate roles[i: 0 .. N - 1].s -> roles[i + 1].r;
3  ⇓
4  par(int i = 0 .. N - 1)
5  context Perm(roles[i].s, 1\2);
6  context Perm(roles[i + 1].r, write);
7  requires I(roles[i], roles[i + 1], roles[i].s)
8  ensures I(roles[i], roles[i + 1], roles[i].r)
9  {
10   roles[i + 1].r = roles[i].s;
11 }
```

(Side condition: senders and receivers do not overlap)

Code-generation for parameterized choreographies

- Surround each parameterized action with an if:
- Is the current endpoint part of the family, and if so, is its `tid` inside the range?
- Then encoded for one endpoint
- Initializing all endpoints, channels, and auxiliary permission/state is annoying but trivial...

  I think

Parameterization: so far

Done:

- Case study distributed summation, distributed leader election
  - Manual encoding of hypothetical VeyMont input, then constructed said hypothetical input.
- Sketch of encoding

To be done:

- Implement choreography verification, code generation
- Port manually encoded case studies as litmus test

Practical issues

Proof-performance things from the manually encoded case study:

- Quantifier loops
- High branching factor
- $+1$/-1 annoyances in quantifiers

Scary open question:

- Quantified stratified permissions: unclear yet what this would look like?

Conclusion

Conclusion

- VeyMont parallelises verified choreographies
- Extended VeyMont with user-specified permissions
- Through additional annotations
- Next: parameterized VeyMont
- Using the `par` block and more annotations

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.

[5] Philip Tasche et al. “Deductive Verification of Parameterized Embedded Systems Modeled in SystemC”. 2024. DOI: 10.1007/978-3-031-50521-8_9.

VerCors & VeyMont

Full parameterized syntax summary

```
1  requires Perm(s.x, 1) ** N >= 2;
2  seq_program Example(Store s, int N) {
3    endpoint[tid: 0 .. N] roles = Role(tid);
4
5    requires
6      (∀ int i = 0 .. N; Perm(s.xs[i], 1));
7    seq_run {
8      communicate
9        role[i: 0 .. N - 1].v <- role[i + 1].v;
10     roles[i: 0 .. N]: s.xs[i] := 10;
11   }
12 }
```