Verified Parameterized Choreographies

Robert Rubbens, Petra van den Bos, Marieke Huisman

June 18th, 2025

Formal Methods & Tools

UNIVERSITY OF TWENTE.

Distributed systems

- Distributed systems are *everywhere*
  - Global internet platforms, embedded settings e.g. cars, software in general
- Distributed systems are *hard*
  - Generic properties: deadlocks, liveness, memory safety, race conditions, ...
  - Highly specific properties
  - Scalability, e.g. increasing number of nodes in a load balancer

Distributed systems

- Distributed systems are *everywhere*
  - Global internet platforms, embedded settings e.g. cars, software in general
- Distributed systems are *hard*
  - Generic properties: deadlocks, liveness, memory safety, race conditions, ...
  - Highly specific properties
  - Scalability, e.g. increasing number of nodes in a load balancer

Establishing correctness of distributed systems is *crucial*

Establishing correctness for distributed systems

- Generic properties, e.g. deadlock freedom
  - Testing, model checking, ...
  - DSL for protocols: *choreographies* [4]
- Highly specific properties: functional correctness verification tool
  - For programs: VerCors, Frama-C, Why3, Verifast, ...
  - For choreographies: *VeyMont* [2, 3, 6]
- Scalability: ...?

Establishing correctness for distributed systems

- Generic properties, e.g. deadlock freedom
  - Testing, model checking, ...
  - DSL for protocols: *choreographies* [4]
- Highly specific properties: functional correctness verification tool
  - For programs: VerCors, Frama-C, Why3, Verifast, ...
  - For choreographies: *VeyMont* [2, 3, 6]
- Scalability: ...?

In this talk:

- Extend *choreographic* verifier & code generator, *VeyMont*
- With support for *parameterized* choreographies
- How?
  - Extend choreographies with parameterization primitives
  - Limitations: one-to-one communication, (kind of) fix the network upfront
  - Encode using "structured parallelism", conditionals, quantifiers

Talk outline

- Choreographies
    - Case study: distributed summation
- Choreographic Verification
    - With VeyMont (powered by VerCors)
- Parameterization in choreographies
- Choreographic projection
- Endpoint projection
- Distributed summation: correctness

Choreographies

Choreographies

- Constrained DSL for protocols[1]
- Essentially:
  - A set of participants: *endpoints*
  - A sequence of interactions: *communications*

[1] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. DOI: 10.1017/9781108981491

Choreographies

- Constrained DSL for protocols[1]
- Essentially:
  - A set of participants: *endpoints*
  - A sequence of interactions: *communications*

```
choreography summation2() {
  endpoint a = Node(int());
  endpoint b = Node(int());
  run {
    communicate a.sum -> b.in;
    communicate b.sum -> a.in;
    a.update();
    b.update();
  }
}
```

[1]Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. DOI: 10.1017/9781108981491

Benefits of choreographies

- Guarantees deadlock-freedom at the message level
  - By virtue of requiring *pairs* of endpoints for communication
- Generate code using the *endpoint projection*, $[\![\bullet]\!]_r$

Conceptual example of endpoint projection

$$\texttt{summation2} \equiv$$

```
choreography summation2() {
  endpoint a = Node(int());
  endpoint b = Node(int());
  run {
    communicate a.sum -> b.in;
    communicate b.sum -> a.in;
    a.update();
    b.update();
  }
}
```

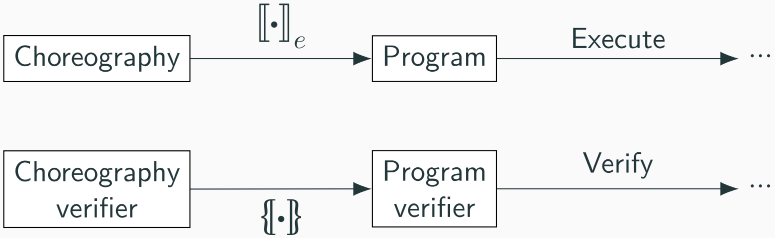
$$[\![\texttt{summation2}]\!]_a \equiv$$

```
void summation2_a(
      Channel to_b,
      Channel from_b) {

  to_b.send(a.sum);
  a.in = from_b.receive();
  a.update();
  /* skip */
}
```

Endpoint projection + verification = …

$$\boxed{\text{Choreography}} \xrightarrow{[\![\cdot]\!]_e} \boxed{\text{Program}} \xrightarrow{\text{Execute}} \cdots$$

Endpoint projection + verification = ...



Choreographic Verification

VeyMont & VerCors

VeyMont:

- Choreographic verifier and code generator
- Support for branching, loops [2, 3] and shared memory [6]
- Built on top of the *VerCors* program verifier
- Verifies choreographies with *pre- and postconditions*
- Functional correctness, memory safety, deadlock freedom

VerCors [5]:

- Verifies concurrent programs with shared memory
- Java, C, OpenCL, CUDA, PVL (Prototypal Verification Language)
- Permission-based separation logic [1]

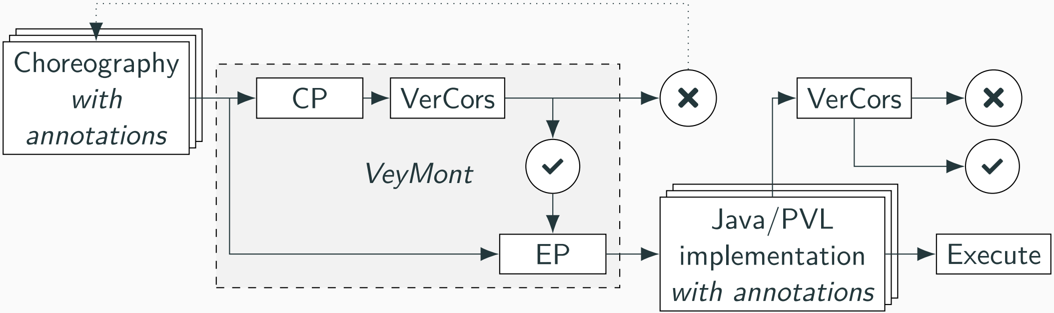
Annotated choreography example

```
choreography summation2() {
  endpoint a = Node(int());
  endpoint b = Node(int());


  ensures a.sum == a.id + b.id; // <--
  ensures b.sum == a.id + b.id;
  run {
    communicate a.sum -> b.in;
    communicate b.sum -> a.in;
    a.update();
    b.update();
  }
}
```

Annotated choreography example

```
choreography summation2() {
  endpoint a = Node(int());
  endpoint b = Node(int());

  requires a.sum == a.id && b.sum == b.id;
  ensures a.sum == a.id + b.id; // <--
  ensures b.sum == a.id + b.id;
  run {
    communicate a.sum -> b.in;
    communicate b.sum -> a.in;
    a.update();
    b.update();
  }
}
```

VeyMont pipeline



- CP: Choreographic Projection, $\{\!|\bullet|\!\}$
- EP: Endpoint Projection, $[\![\bullet]\!]_r$

summation2: choreographic projection $\llbracket\cdot\rrbracket$

```
choreography summation2() {
  endpoint a = Node(int());
  endpoint b = Node(int());

  requires a.sum == a.id;
  requires b.sum == b.id;
  ensures a.sum == a.id + b.id;
  ensures b.sum == a.id + b.id;
  run {
    communicate a.sum -> b.in;
    communicate b.sum -> a.in;
    a.update();
    b.update();
  }
}
```

```
void summation2_cp() {
  Node a = new Node(int());
  Node b = new Node(int());

  assert a.sum == a.id
  assert b.sum == b.id;

  b.in = a.sum;
  a.in = b.sum;

  a.update();
  b.update();

  assert a.sum == a.id + b.id;
  assert b.sum == a.id + b.id;
}
```

Parameterization in choreographies

Parameterization in choreographies

Data parameterization:

```
choreography summationN(int msg) {
  endpoint s = Sender(msg);
  endpoint r = Receiver();
  run {
    communicate s.msg -> r.inbox;
  }
}
```

- Limitation: numbers of endpoints statically known
- Why: natural scalability
  - Sharding, ring networks, users, etc.

Parameterization in choreographies

Data parameterization:

```
choreography summationN(int msg) {
  endpoint s = Sender(msg);
  endpoint r = Receiver();
  run {
    communicate s.msg -> r.inbox;
  }
}
```

- Limitation: numbers of endpoints statically known
- Why: natural scalability
  - Sharding, ring networks, users, etc.

Parameterization in *number of endpoints*

Parameterization primitives

Endpoint family:

```
endpoint ns[i := 0 .. N] = Node(i, int());
```

Endpoint quantification:

```
(\endpoints ns[i := 0..N]; nodes[i].n < N-1)
```

Parameterized communicate:

```
communicate ns[i := 0..N-1].sum -> ns[i+1].in;
```

Parameterization primitives

Endpoint family:

```
endpoint ns[i := 0 .. N] = Node(i, int());
```

Endpoint quantification:

```
(\endpoints ns[i := 0..N]; nodes[i].n < N-1)
```

Parameterized communicate:

```
communicate ns[i := 0..N-1].sum -> ns[i+1].in;
```

Limitations:

1. In `ns[`E`]`, E cannot read the heap
   - `ns[store.x]` ✗
2. E must be invertible
   - `i + 1` $\iff$ `i - 1`

summation2 → summationN

```
endpoint ns[i := 0 .. N] = Node(i, int());

run:
while ((\endpoints ns[i := 0..N]; nodes[i].n < N-1)) {
  communicate ns[i := 0..N-1].sum -> ns[i+1].in;
  communicate ns[N-1].sum -> ns[0].in;
  nodes[i := 0..N].update();
}
```

Projections

Choreographic projection: endpoints

$\{\![\bullet]\!\}$

Endpoint family:

$\{\![$`endpoint ns[i := 0 .. N] = Node(i, int());`$]\!\} =$

```
seq<Node> ns = [];
for (int i = 0 .. N) {
  ns = ns + [new Node()];
}
```

Choreographic projection: endpoints

Endpoint family:

```
⦃endpoint ns[i := 0 .. N] = Node(i, int());⦄ =
        seq<Node> ns = [];
        for (int i = 0 .. N) {
          ns = ns + [new Node()];
        }
```

Quantification:

```
⦃(\endpoints ns[i := 0..N]; nodes[i].n < N-1)⦄ =
    (\forall int i = 0..N; nodes[i].n < N-1)
```

Choreographic projection: `communicate` (*)

$\llbracket$`communicate ns[i := 0..N-1].sum -> ns[i+1].in;`$\rrbracket =$

```
// Start N threads, and wait until finished
par (int i = 0 .. N - 1)
// That can read ns[i].sum
requires Perm(ns[i].sum, read);
// And write to ns[i + 1].in
requires Perm(ns[i + 1].in, write);
{
  // Re-use regular encoding
  ⟦communicate ns[i].sum -> ns[i + 1].in;⟧
}
```

Endpoint projection: endpoints

$$[\![\ldots]\!]_{\texttt{ns[}i\texttt{]}}$$

Endpoint family:

$$[\![\texttt{endpoint ns[i := 0 .. N] = Node(i, int());}]\!]_{\texttt{ns[}i\texttt{]}} =$$

```
void summationN_nodes_i(seq<Node> ns, int i) {
  [[run { ... }]]ns[i]
}
```

Endpoint projection: endpoints

$$[\![\dots]\!]_{\texttt{ns[}i\texttt{]}}$$

Endpoint family:

$$[\![\texttt{endpoint ns[i := 0 .. N] = Node(i, int());}]\!]_{\texttt{ns[}i\texttt{]}} =$$

```
void summationN_nodes_i(seq<Node> ns, int i) {
  [[run { ... }]]ns[i]
}
```

Endpoint quantification:

$$[\![\texttt{(\textbackslash endpoints ns[k := 0..N]; nodes[k].n < N-1)}]\!]_{\texttt{ns[}i\texttt{]}} =$$

$$\texttt{0 <= } i \texttt{ \&\& } i \texttt{ < N ==> nodes[}i\texttt{].n < N-1}$$

Endpoint projection: parameterized `communicate`

$$[\![\ldots]\!]_{\texttt{ns[}i\texttt{]}}$$

$[\![$`communicate ns[i := 0..N-1].sum -> ns[i+1].in;`$]\!]_{\texttt{ns[}i\texttt{]}} =$

```
// i in sending range
if (0 <= i && i < N - 1) {
  ns_ns[i].writeValue(ns[i].sum)
}
// i in receiving range
if (0 + 1 <= i && i < N - 1 + 1) {
  ns[i].in = ns_ns[i - 1].readValue();
}
```

`summationN`: Functional correctness

summationN correctness

```
choreography summationN(int N) {
  endpoint ns[i := 0 .. N] = Node(i, int());
  ensures POST;
  run {
    loop_invariant INV;
    while ((\endpoints ns[i := 0..N]; ns[i].n < N-1)) {
      communicate ns[i := 0..N-1].sum -> ns[i+1].in;
      communicate ns[N-1].sum -> ns[0].in;
      nodes[i := 0..N].update();
    }
    // Proof step
    endpoint nodes[i := 0..N].generalizeTo0();
  }
}
```

POST $\equiv$

```
(\endpoints ns[i := 0..N];
    ns[i].sum == sum(ns, 0, ns[i].n));
```

Conclusion

Conclusion

- Extended choreographies with primitives for parameterization
- Two limitations:
  1. Family indexing must be heap-independent
  2. Indexing must be one-to-one
- Verification: sequences & `par` blocks
- Code generation: conditionals (implication and `if`)
- Manual encoding of *distributed summation*

Robert Rubbens
Formal Methods & Tools
University of Twente
r.b.rubbens@utwente.nl - `https://bobrubbens.nl`

References

[1] Afshin Amighi et al. "Permission-Based Separation Logic for Multithreaded Java Programs". In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:2)2015.

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

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

[4] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. DOI: 10.1017/9781108981491.

[5] Lukas Armborst et al. "The VerCors Verifier: A Progress Report". In: *CAV*. 2024. DOI: 10.1007/978-3-031-65630-9_1.

[6] Robert Rubbens, Petra van den Bos, and Marieke Huisman. "VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory". In: *iFM*. 2024. DOI: 10.1007/978-3-031-76554-4_12.

Bonus

(*)

- Deadlock freedom: branch unanimity
- Communicate: injectivity of E
- Verification only requires injectivity, not invertibility

`summationN`:

- Ghost code for sending ghost memory back and forth
- Proof steps for generalizing correctness result of one individual endpoint

summationN code

```
choreography summationN(int N) {
  endpoint nodes[i := 0 .. N] = Node(i, int());
  run {
    while ((\endpoints nodes[i := 0..N]; nodes[i].n < N-1)) {
      communicate nodes[i := 0..N-1].sum -> nodes[i+1].in;
      communicate nodes[N-1].sum -> nodes[0].in;
      nodes[i := 0..N].update();
    }
  }
}
```

INV

```
(\endpoints ns[i := 0..N];
    ns[i].total ==
        sum(ns, ns[i].i, ns[i].n))
```