# Safety & Performance in Generated Coordination Code

Christopher Esterhuyse

# Contents

- Background
  - Coordination
  - Reo
  - Reo Compiler
  - Runtime Coordination
- Contribution Overview
- Code Generation
  - Rust Language
  - Translation Pipeline
  - Example
- Generated Object Behavior
  - Execution
  - Performance
  - Safety
- Protocol Adherence
  - Problem Explained
  - Static Checking
  - Example

# Contents

- Background
  - Coordination
  - Reo
  - Reo Compiler
  - Runtime Coordination
- Contribution Overview
- Code Generation
  - Rust Language
  - Translation Pipeline
  - Example
- Generated Object Behavior
  - Execution
  - Performance
  - Safety
- Protocol Adherence
  - Problem Explained
  - Static Checking
  - Example

# Background ●○○ Coordination

- In concurrent programming: Coordination involves **interactions** between actors.

# Background ●○○ Coordination

- In concurrent programming: Coordination involves **interactions** between actors.
- **Imperative** languages express sequences of actions, mutating the program state.
- Concurrent programs must mix code for **computation** and **coordination**. Results in coupling.

```
void work_x(int* msg) {
  for(int i = 0;; i++) {
    sem_wait(&b);
    *msg = i;
    sem_post(&a);
  }
}

void work_y(int* msg) {
  for(;;) {
    sem_wait(&a);
    printf(“%d\n”, *msg);
    sem_post(&b);
  }
}
```

# Background ●○○ Reo

**Coordination language**: Express coordination in a system more abstractly.

- Reo language is **graphical**: defines a *connector* with nodes and relations (≈hyperedges).
- Relations constrain how nodes interact, defining how data can **flow**.

Eg: primitive *sync* in graphical syntax:

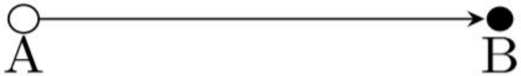


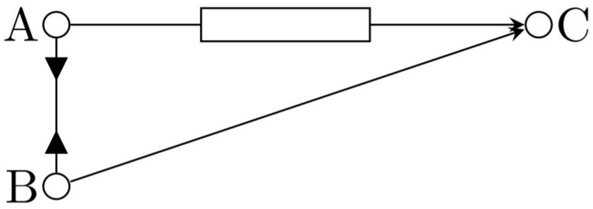
Image source: “Reasoning about connectors using Coq and Z3”

# Background ●○○ Reo

**Coordination language**: Express coordination in a system more abstractly.

- Reo language is **graphical**: defines a *connector* with nodes and relations (≈hyperedges).
- Relations constrain how nodes interact, defining how data can **flow**.
- Nodes exposed to the environment are **ports**.
- Non-primitive connectors are **built** from others by exposing or coupling ports.

Eg: *alternator2* in graphical & textual syntax:

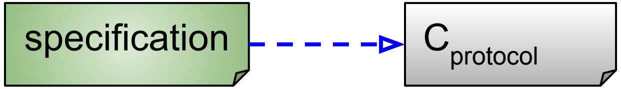


```
alternator2(A, B, C){
    syncdrain(A, B)
    fifo1(A, C)
    sync(B, C)
}
```

Image source: “Reasoning about connectors using Coq and Z3”

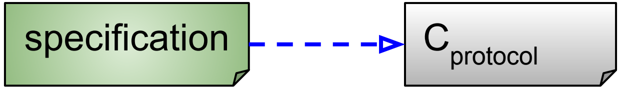
# Background ●●○ Reo Compiler

- Tool for translating a Reo specification to a *protocol object* in a target language.



# Background ●●○ Reo Compiler

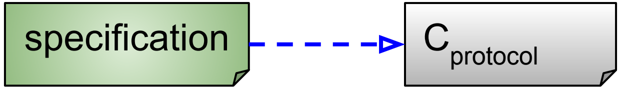
- Tool for translating a Reo specification to a *protocol object* in a target language.



- Has *backends* for targeting Java, Maude, Promela, …
  - The role of the protocol object depends on the language.
  - For Java: generate coordinating Java glue-code.

# Background ●●○ Reo Compiler

- Tool for translating a Reo specification to a *protocol object* in a target language.

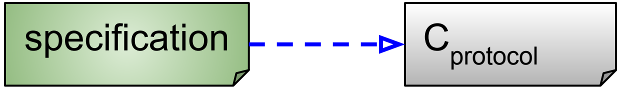


- Has *backends* for targeting Java, Maude, Promela, …
  - The role of the protocol object depends on the language.
  - For Java: generate coordinating Java glue-code.

```
void work_x(int* msg) {
  for(int i = 0;; i++) {
    sem_wait(&b);
    *msg = i;
    sem_post(&a);
  }
}
```

# Background ●●○ Reo Compiler

- Tool for translating a Reo specification to a *protocol object* in a target language.



- Has *backends* for targeting Java, Maude, Promela, …
  - The role of the protocol object depends on the language.
  - For Java: generate coordinating Java glue-code.

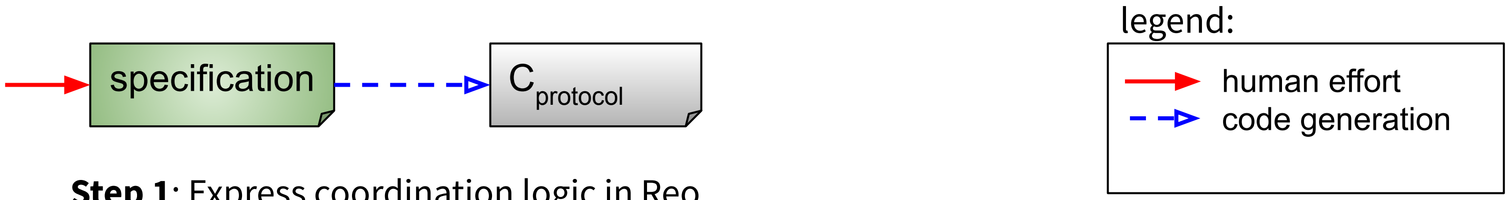
- Developed at CWI. Has seen work by…
  - Sung-Shik Jongmans
  - Kasper Dokter
  - Benjamin Lion
  - …

# Background ●●○ Reo Compiler

Building a coordinating program with the Reo compiler:

# Background ●●○ Reo Compiler

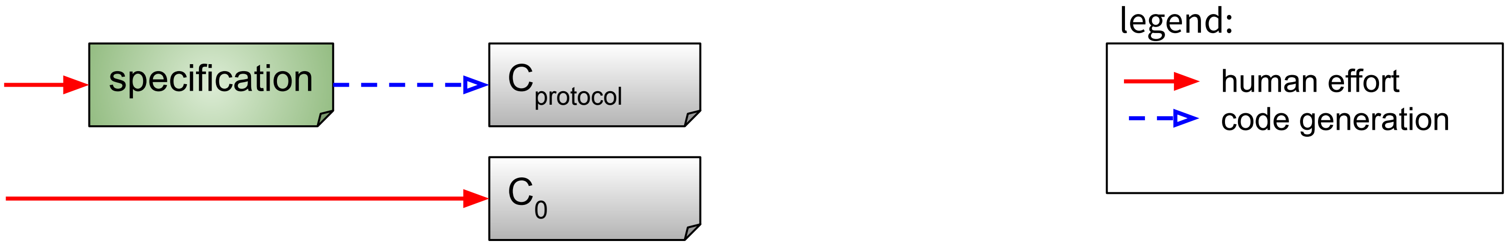
Building a coordinating program with the Reo compiler:



**Step 1**: Express coordination logic in Reo. Compile to protocol component(s).

# Background ●●○ Reo Compiler

Building a coordinating program with the Reo compiler:

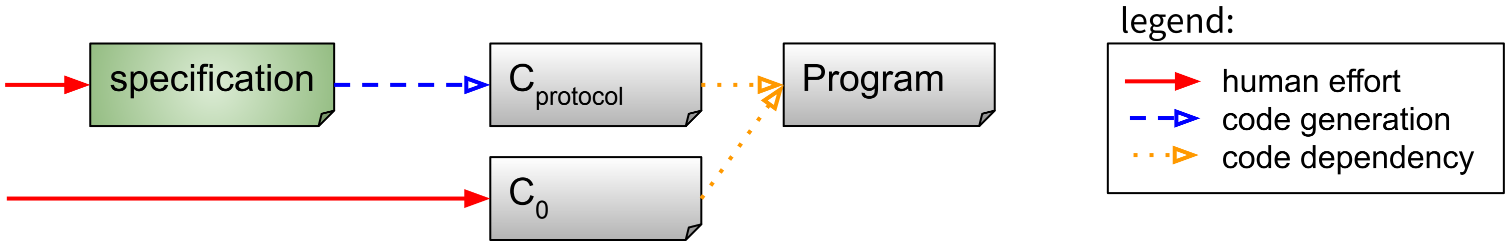


**Step 2:** Express computation logic as components. Data exchange is abstracted to port operations with the environment.

```
void work_x(port p) {
  for(int i = 0;; i++) {
    p.put(i);
  }
}
```

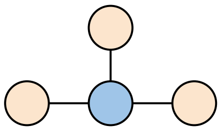
# Background ●●○ Reo Compiler

Building a coordinating program with the Reo compiler:



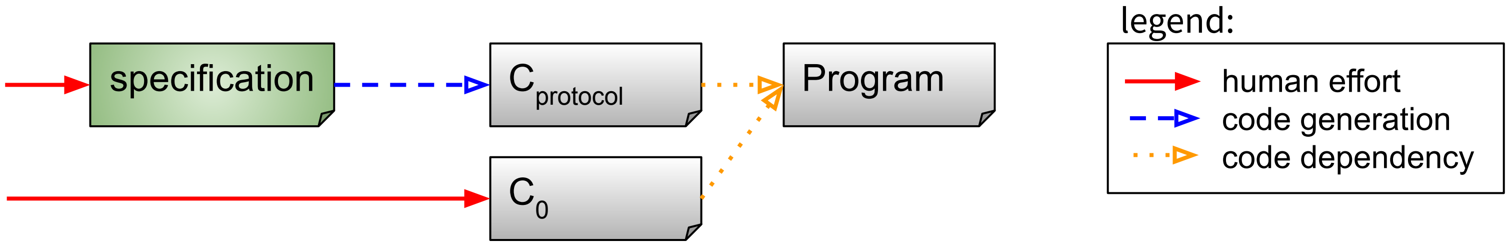
**Step 3**: Build program by linking components’ ports as specified.

Eg: protocol coordinating 3 other components



# Background ●●○ Reo Compiler

Building a coordinating program with the Reo compiler:

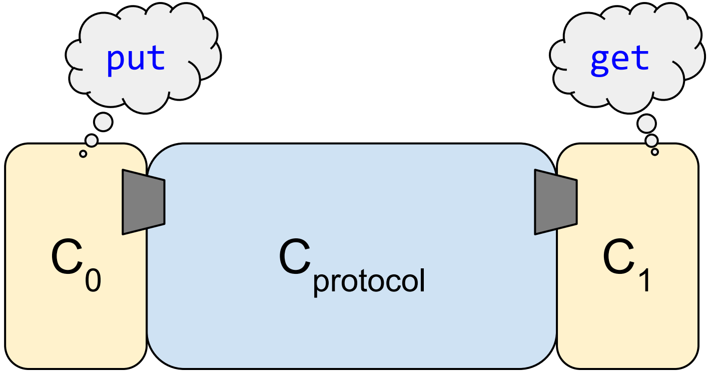


Advantages:

- **Direct:** Modify and check protocol via specification.
- **Indirect:** Program components are modular, reusable, maintainable.

# Background ●●● Runtime Coordination

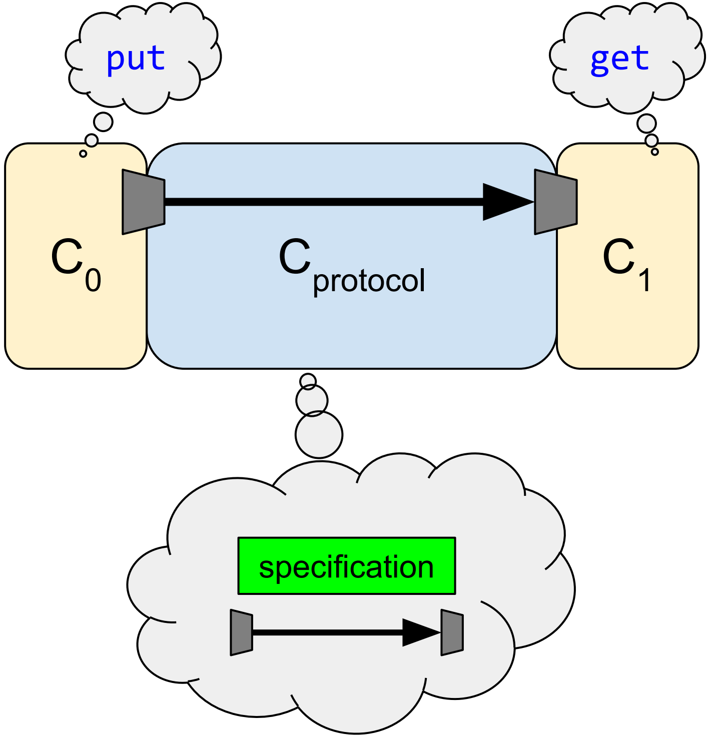
- User components initiate put or get actions on their local ports as they like.



# Background ●●● Runtime Coordination

- User components initiate put or get actions on their local ports as they like.

- *Protocol object* completes sets of actions as soon as they can be organized into a specified *interaction.*



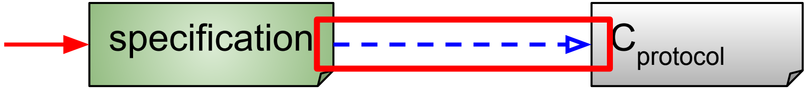
# Contents

- Background
  - Coordination
  - Reo
  - Reo Compiler
  - Runtime Coordination
- Contribution Overview
- Code Generation
  - Rust Language
  - Translation Pipeline
  - Example
- Generated Object Behavior
  - Execution
  - Performance
  - Safety
- Protocol Adherence
  - Problem Explained
  - Static Checking
  - Example

# Contribution Overview

**Task**: Make a new Reo compiler backend for a systems-language target.

**Holy grail**: Reo-generated code is as safe and performant as hand-crafted code.



| Make it safe | Make it practical |
|---|---|
| Correct wrt. **specification**: protocol and general Reo semantics preserved. | Implement **optimizations** (eg. reference-passing port operations in shared memory). |
| Correct wrt. **target language**: code type-checks, no undefined behavior, etc. | Make it possible for programmers to reason about **liveness**. |

# Contents

- Background
  - Coordination
  - Reo
  - Reo Compiler
  - Runtime Coordination
- Contribution Overview
- Code Generation
  - Rust Language
  - Translation Pipeline
  - Example
- Generated Object Behavior
  - Execution
  - Performance
  - Safety
- Protocol Adherence
  - Problem Explained
  - Static Checking
  - Example

# Code Generation ●○○ Rust language

We compile to Rust.

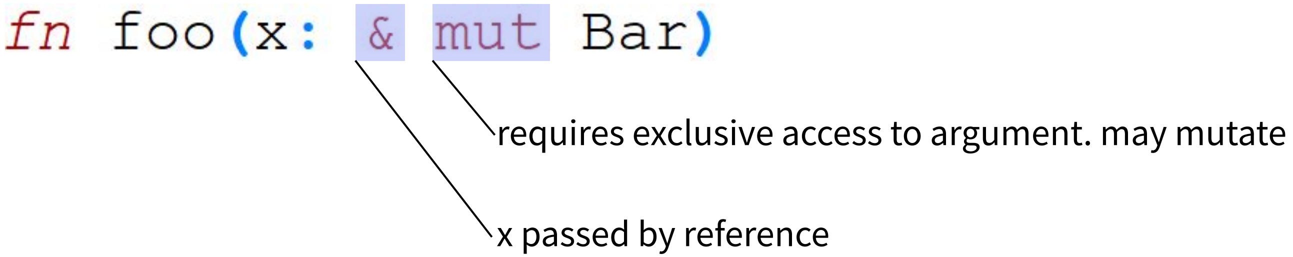
- Systems language
  - Low-level resource manipulations → optimizations.
  - Cheap interoperability with C++ and C → larger audience.

# Code Generation ●○○ Rust language

We compile to Rust.

- Systems language
  - Low-level resource manipulations → optimizations.
  - Cheap interoperability with C++ and C → larger audience.

- Expressive affine type system
  - Expressive component and port API to statically enforce safety.



```
fn foo(x: & mut Bar)
```

requires exclusive access to argument. may mutate

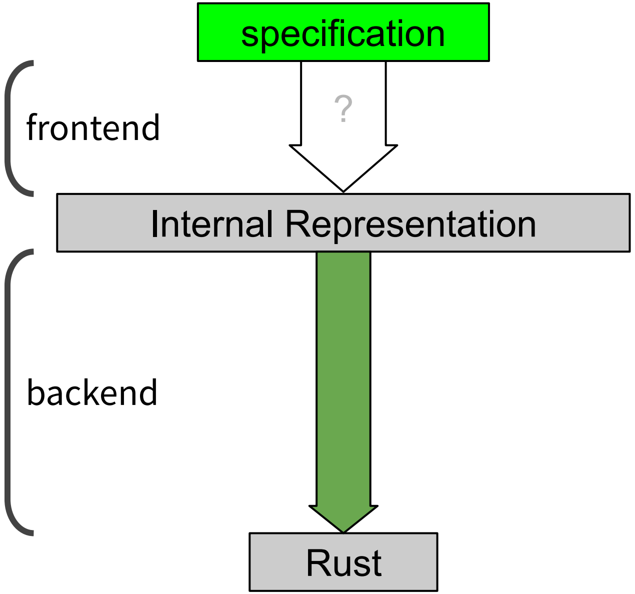
x passed by reference

# Code Generation ●○○ Rust language

We compile to Rust.

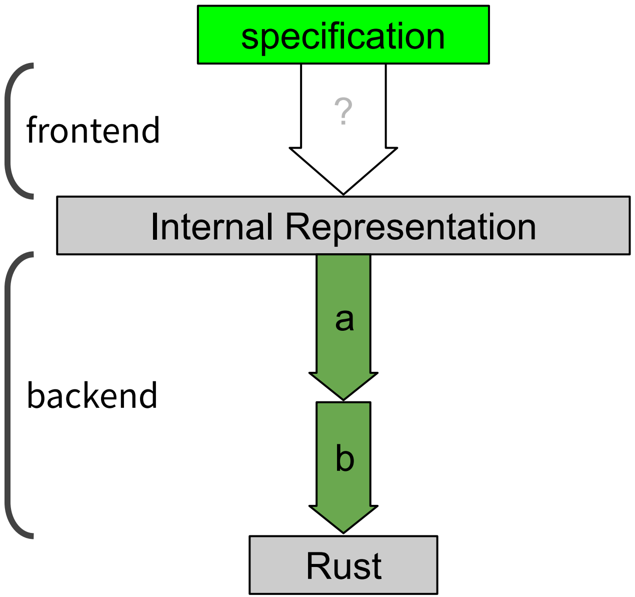
- Systems language
  - Low-level resource manipulations → optimizations.
  - Cheap interoperability with C++ and C → larger audience.
- Expressive affine type system
  - Expressive component and port API to statically enforce safety.
- Popular modern language
  - Lively, growing community → likely to stay a useful Reo target.
  - Modern features for safety and productivity (eg. closures, matching).

# Code Generation ●●○ Translation Pipeline



# Code Generation ●●○ Translation Pipeline

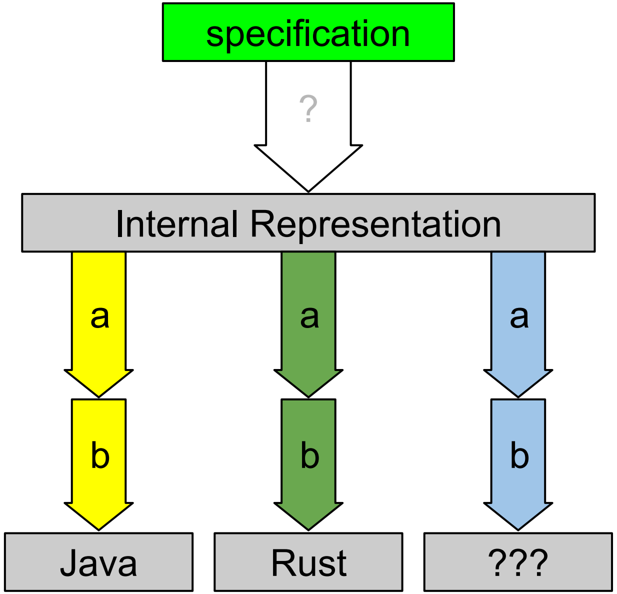
- Back-end translation can be broken into stages:
    a. **Abstract**: Port data types resolved, interactions laid out as sequences of actions.
    b. **Concrete**: Emit executable, concrete Rust.



# Code Generation ●●○ Translation Pipeline

- Back-end translation can be broken into stages:
  a. **Abstract**: Port data types resolved, interactions laid out as sequences of actions.
  b. **Concrete**: Emit executable, concrete Rust.

- Similar for all imperative languages.

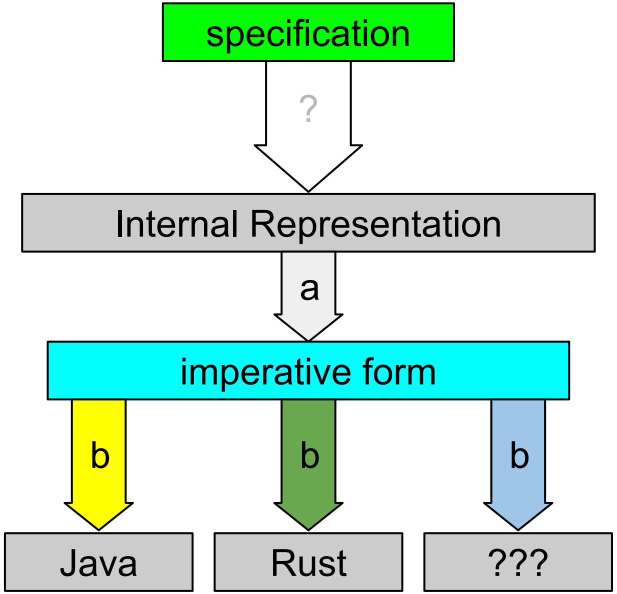


# Code Generation ●●○ Translation Pipeline

- Back-end translation can be broken into stages:
  a. **Abstract**: Port data types resolved, interactions laid out as sequences of actions.
  b. **Concrete**: Emit executable, concrete Rust.

- Similar for all imperative languages.

- Introduce *imperative form* between **a** and **b**.
  a. Sensible for many imperative targets (eg. not promela).
  b. Reo compiler does **a**. Rust does **b**.
  → Reo and Rust decoupled.



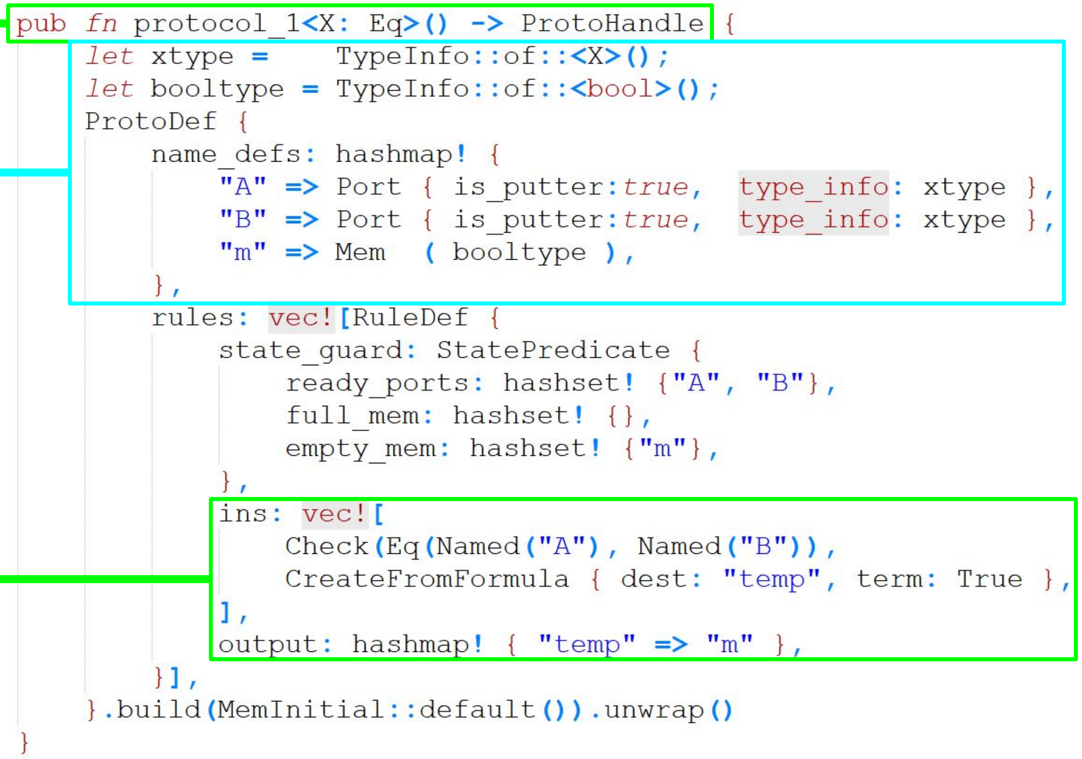
**Note**: only Rust currently does this.

# Code Generation ●●● Example

User sees

Explicit data types

Interaction as action sequence



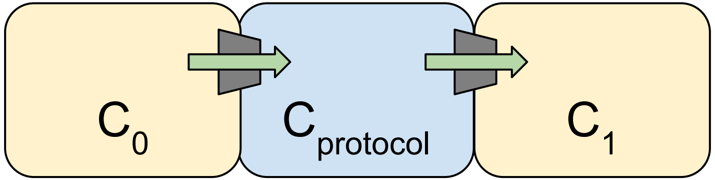
```
pub fn protocol_1<X: Eq>() -> ProtoHandle {
    let xtype =    TypeInfo::of::<X>();
    let booltype = TypeInfo::of::<bool>();
    ProtoDef {
        name_defs: hashmap! {
            "A" => Port { is_putter:true,  type_info: xtype },
            "B" => Port { is_putter:true,  type_info: xtype },
            "m" => Mem  ( booltype ),
        },
        rules: vec![RuleDef {
            state_guard: StatePredicate {
                ready_ports: hashset! {"A", "B"},
                full_mem: hashset! {},
                empty_mem: hashset! {"m"},
            },
            ins: vec![
                Check(Eq(Named("A"), Named("B"))),
                CreateFromFormula { dest: "temp", term: True },
            ],
            output: hashmap! { "temp" => "m" },
        }],
    }.build(MemInitial::default()).unwrap()
}
```

# Contents

- Background
  - Coordination
  - Reo
  - Reo Compiler
  - Runtime Coordination
- Contribution Overview
- Code Generation
  - Rust Language
  - Translation Pipeline
  - Example
- Generated Object Behavior
  - Execution
  - Performance
  - Safety
- Protocol Adherence
  - Problem Explained
  - Static Checking
  - Example

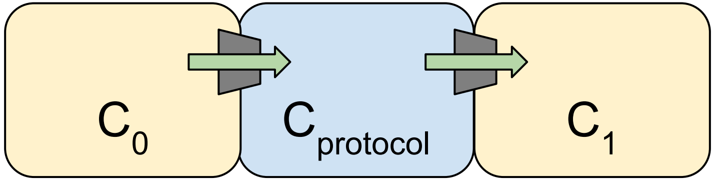
# Generated Object Behavior ●○○ Execution

- Reo→Java:
  - Distinct **class**. Implements `Protocol` interface.
  - Protocol is threaded.
  - Data exchanged at ports.
  - Value’s reference is moved.

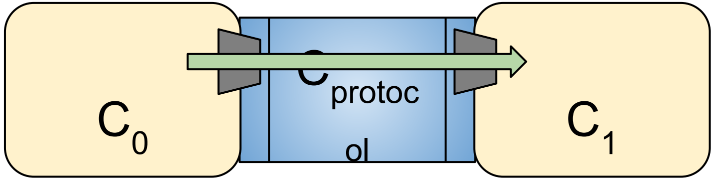


# Generated Object Behavior ●○○ Execution

- Reo→Java:
  - Distinct **class**. Implements `Protocol` interface.
  - Protocol is threaded.
  - Data exchanged at ports.
  - Value’s reference is moved.

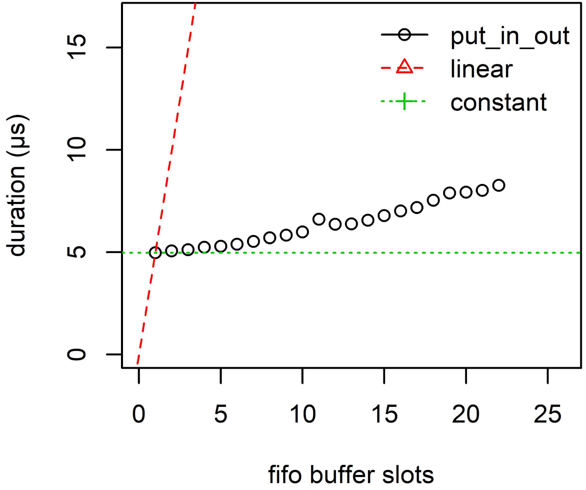


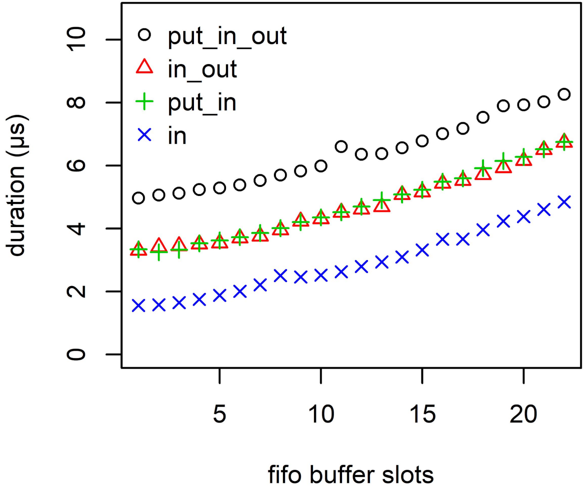
- Reo→Rust:
  - Distinct **function**. Builds `Protocol` instance.
  - Protocol is data. Boundary components do work.
  - Data acquired from the source (eg. $C_0 \to C_1$).
  - Value is moved.



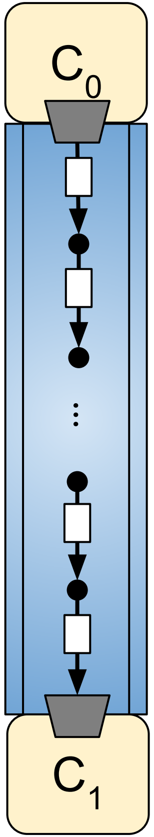
# Generated Object Behavior ●●○ Performance

- Optimization 1: internal reference-passing and aliasing
  - Values moved *within* protocol object by reference.
  - Cost of internal move or replication is a small constant.



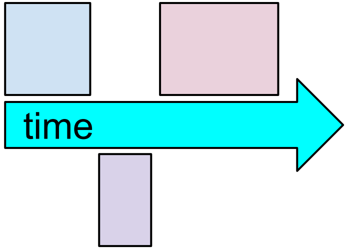


fifo-N connector:



# Generated Object Behavior ●●○ Performance

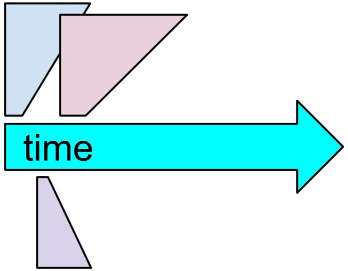
- Naïvely, interactions occur sequentially
  - Interactions influence protocol's state.
    → they must be serializable



# Generated Object Behavior ●●○ Performance

- Naïvely, interactions occur sequentially
  - Interactions influence protocol’s state.
    → they must be serializable

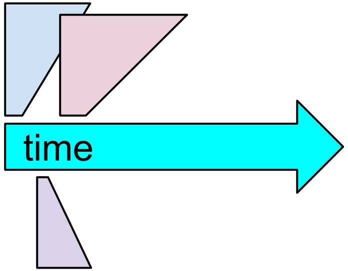
- Optimization 2 : concurrent data movements
  - Interactions are still serialized, but their work may overlap



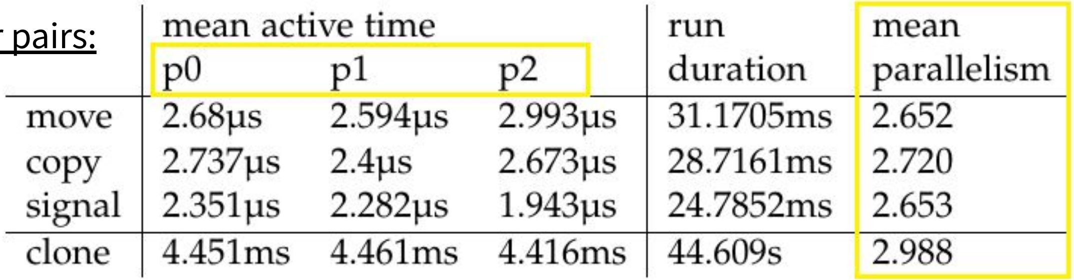
# Generated Object Behavior ●●○ Performance

- Naïvely, interactions occur sequentially
  - Interactions influence protocol’s state.
    → they must be serializable

- Optimization 2 : concurrent data movements
  - Interactions are still serialized, but their work may overlap



3 putter-getter pairs:

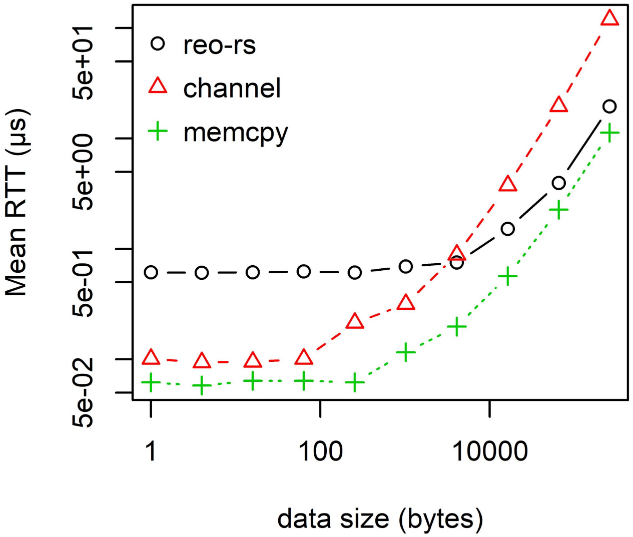


| | mean active time | | | run duration | mean parallelism |
|---|---|---|---|---|---|
| | p0 | p1 | p2 | | |
| move | 2.68μs | 2.594μs | 2.993μs | 31.1705ms | 2.652 |
| copy | 2.737μs | 2.4μs | 2.673μs | 28.7161ms | 2.720 |
| signal | 2.351μs | 2.282μs | 1.943μs | 24.7852ms | 2.653 |
| clone | 4.451ms | 4.461ms | 4.416ms | 44.609s | 2.988 |

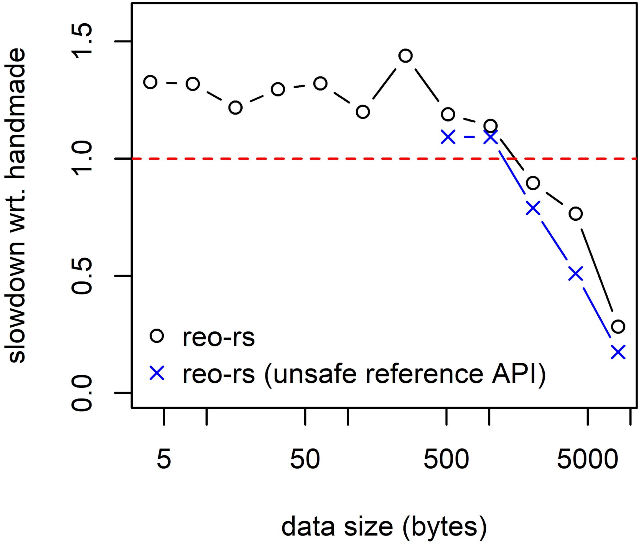
# Generated Object Behavior ●●○ Performance

Competitive vs. hand-crafted code for non-trivial protocols.

1-thread fifo1:



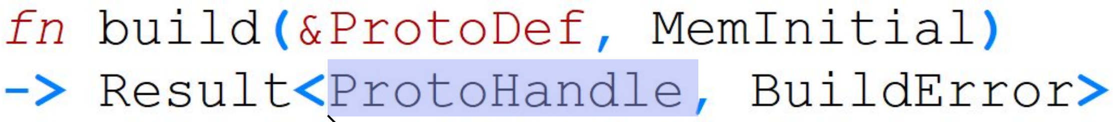
alternator2:



# Generated Object Behavior ●●● Safety

Port & protocol objects are always in valid state.

- Protocol objects only acquired with `build`, which ensures...
  - Initialization is beyond user’s control (ie. port linkage and initial state)



```
fn build(&ProtoDef, MemInitial)
-> Result<ProtoHandle, BuildError>
```

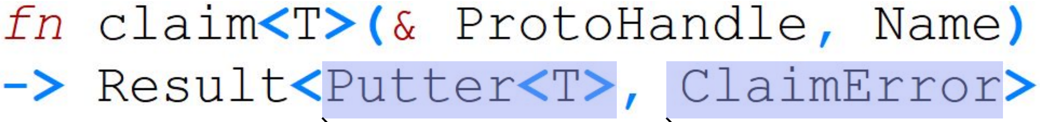
returned object is initialized

# Generated Object Behavior ●●● Safety

Port & protocol objects are always in valid state.

- Protocol objects only acquired with `build`, which ensures…
  - Initialization is beyond user’s control (ie. port linkage and initial state)

- Ports acquired only acquired from a protocol with `claim`, which ensures...
  - port objects are of the correct type (data type, orientation).
  - cannot duplicate logical ports.



```
fn claim<T>(& ProtoHandle, Name)
-> Result<Putter<T>, ClaimError>
```

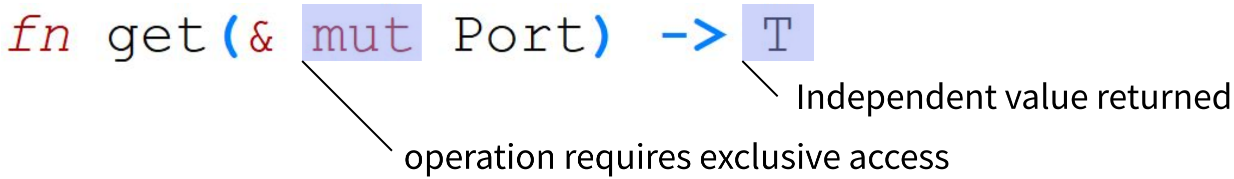
may return error

port data type and orientation is fixed

# Generated Object Behavior ●●● Safety

Reo’s semantics are preserved.

- Port `put` and `get` use Rust’s ownership semantics to ensure...
  - values are passed as expected.
  - a port cannot do two things at once.



```
fn get(& mut Port) -> T
```

Independent value returned

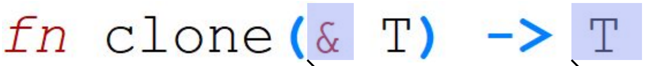
operation requires exclusive access

# Generated Object Behavior ●●● Safety

Reo’s semantics are preserved.

- Port `put` and `get` use Rust’s ownership semantics to ensure...
  - values are passed as expected.
  - a port cannot do two things at once.

- Replication via user-defined `clone` operation.
  - Signature prohibits side effects → more optimizations possible.



```
fn clone(& T) -> T
```

independent value returned

original is immutable

# Contents

- Background
  - Coordination
  - Reo
  - Reo Compiler
  - Runtime Coordination
- Contribution Overview
- Code Generation
  - Rust Language
  - Translation Pipeline
  - Example
- Generated Object Behavior
  - Execution
  - Performance
  - Safety
- Protocol Adherence
  - Problem Explained
  - Static Checking
  - Example

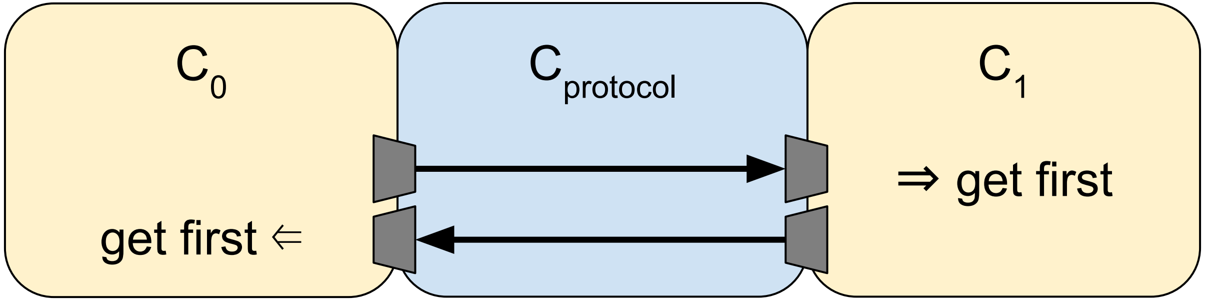
# Protocol Adherence ●○○ Problem Explained

- The behavior of a system is the *composition* of the behavior of its components. System behavior is constrained by the protocol.

...

# Protocol Adherence ●○○ Problem Explained

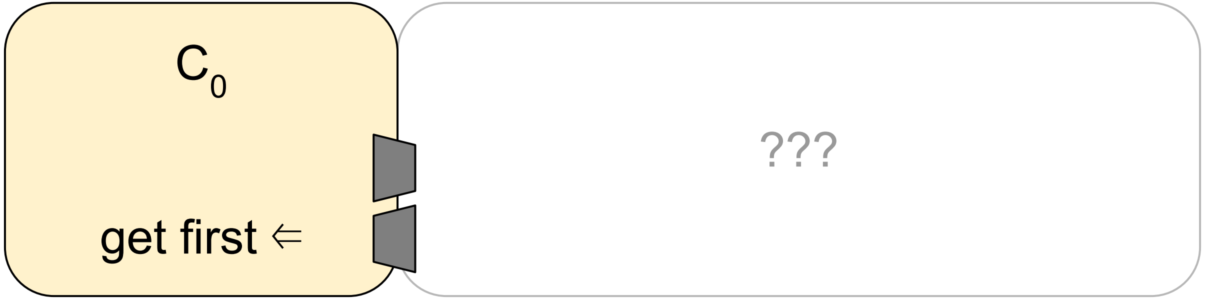
- The behavior of a system is the *composition* of the behavior of its components. System behavior is constrained by the protocol.



- Eg. this system is specified to have no behavior.

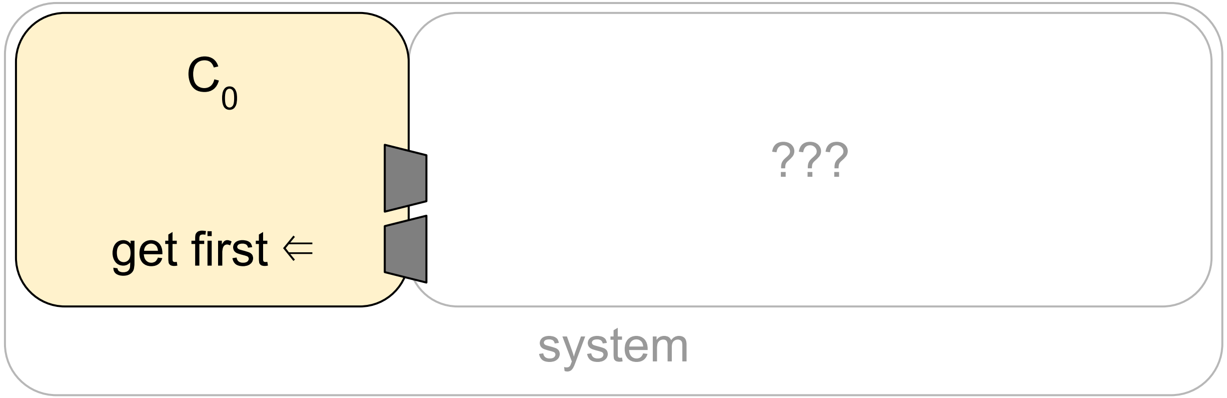
# Protocol Adherence ●○○ Problem Explained

- Recall: Components are ignorant of their environment.



# Protocol Adherence ●○○ Problem Explained

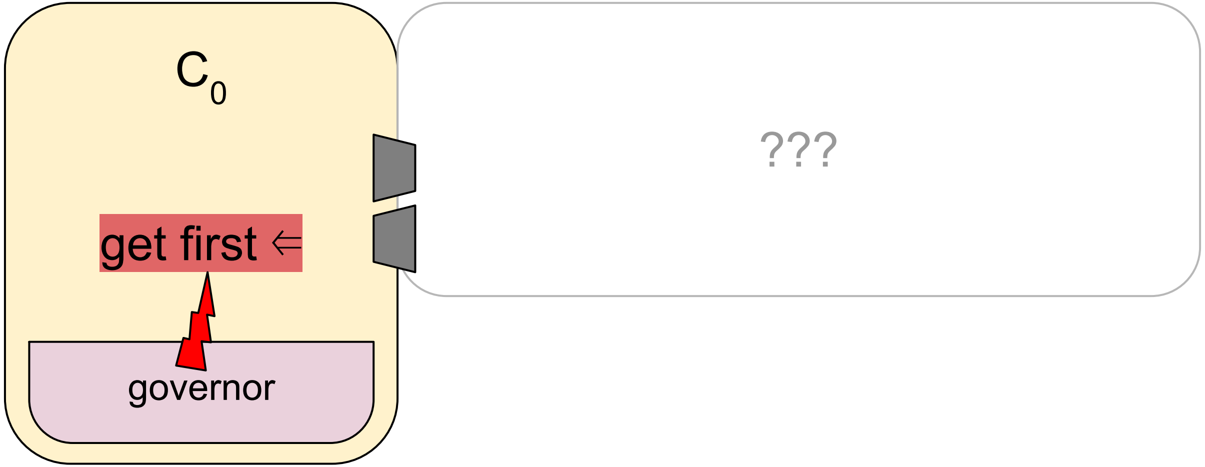
- Recall: Components are ignorant of their environment.



- Components define **local** behavior, but we observe **global** behavior at runtime.
  - We don't know the effect of local actions on the system's *liveness*.

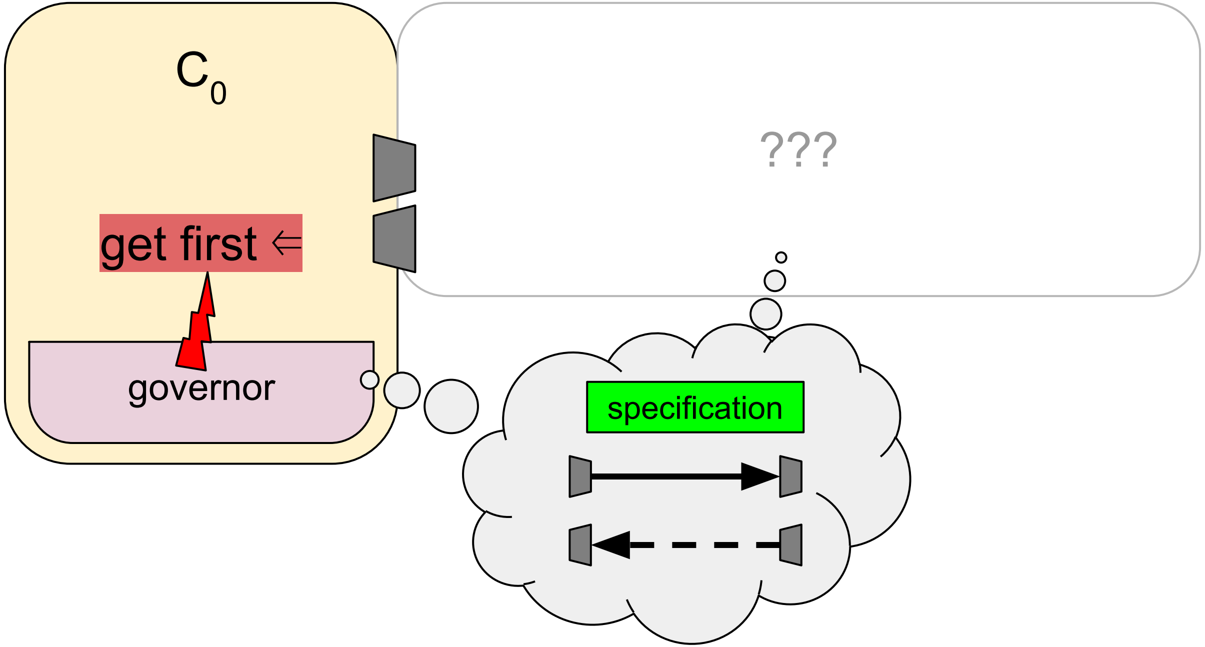
# Protocol Adherence ●●○ Static Checking

- **Governor:** Enforces *adherence*, i.e., local actions don’t add new constraints.



# Protocol Adherence ●●○ Static Checking

- **Governor:** Enforces *adherence*, i.e., local actions don't add new constraints.
  - constructed from the specification of the connected protocol (but has a localized view).



# Protocol Adherence ●●○ Static Checking

- We focus on **static** governors. Deviation is checked at compile-time.

# Protocol Adherence ●●○ Static Checking

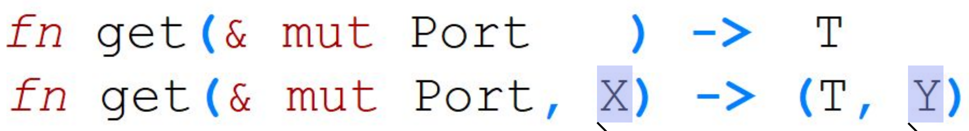
- We focus on **static** governors. Deviation is checked at compile-time.

- Encode governors into Rust’s type system.
  - Component is adherent ⇔ Rust code compiles.
  - **Why:** Programmer needs the Rust compiler anyway. Governor is only a Rust dependency.
  - **How:** Translate “port operation is permitted in current state” to “variable has correct type”

# Protocol Adherence ●●○ Static Checking

- We focus on **static** governors. Deviation is checked at compile-time.

- Encode governors into Rust’s type system.
  - Component is adherent ⇔ Rust code compiles.
  - **Why:** Programmer needs the Rust compiler anyway. Governor is only a Rust dependency.
  - **How:** Translate “port operation is permitted in current state” to “variable has correct type”:
    - component always has one *token.* Its type encodes the protocol’s state.
    - port operations replace the token ⇔ protocol’s state is changed.



before:

```
fn get(& mut Port   ) ->  T
```

after:

```
fn get(& mut Port, X) -> (T, Y)
```

Initialized value returned

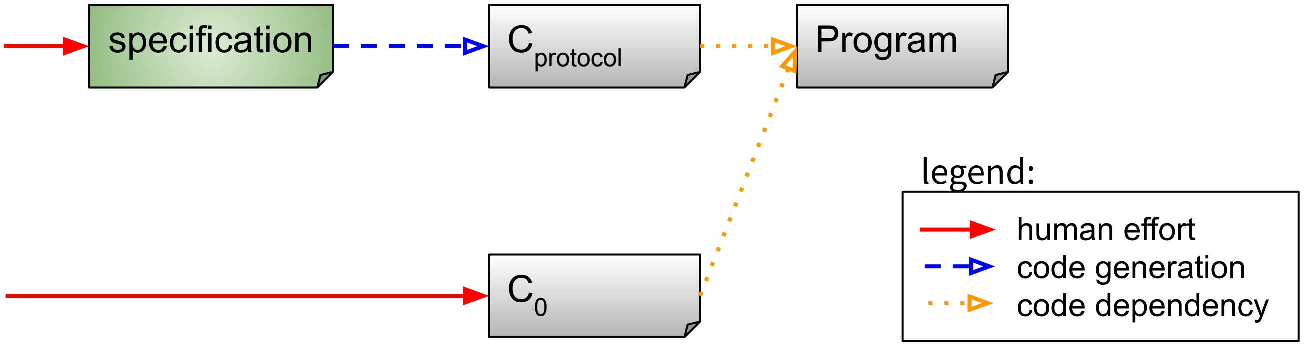
argument consumed

...

# Protocol Adherence ●●○ Static Checking

Governors supplement the user's workflow:

before...



specification

$C_{protocol}$

Program

$C_0$

legend:

human effort

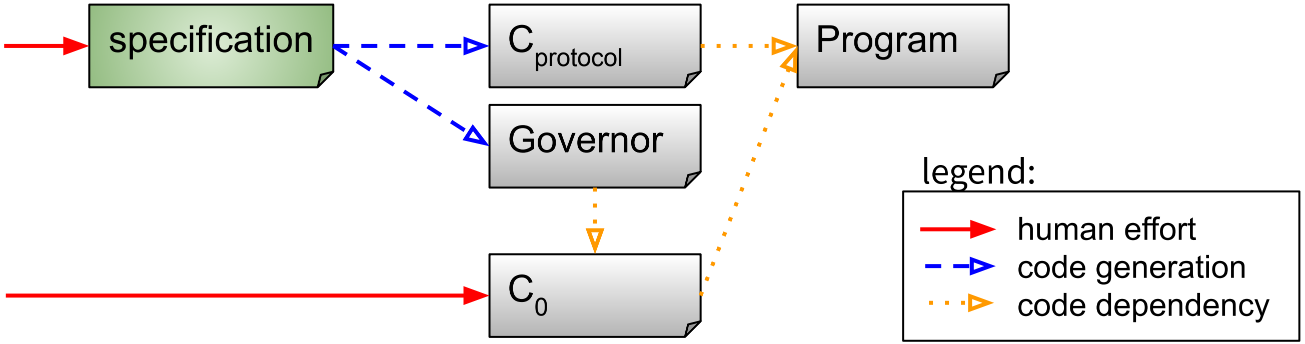
code generation

code dependency

# Protocol Adherence ●●○ Static Checking

Governors supplement the user's workflow:

… after



# Protocol Adherence ●●● Example

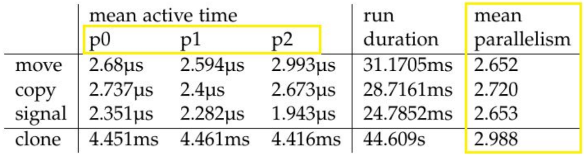
```
fn unsafe_component_0(port_a, port_b) {
     DATA = port_a.get();
     port_b.put(DATA);
     port_b.put(DATA); // (deadlocks at runtime)
}
```

# Protocol Adherence ●●● Example

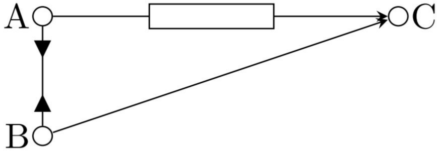
```
fn unsafe_component_0(port_a, port_b) {
    DATA = port_a.get();
    port_b.put(DATA);
    port_b.put(DATA); // (deadlocks at runtime)
}


fn __safe_component_0(t, port_a, port_b) {
    (t, DATA) = port_a.get(t);
    t = port_b.put(t, DATA);
    t = port_b.put(t, DATA);
    // TYPE ERROR  ^-- Expected FOO not BAR.
}
```

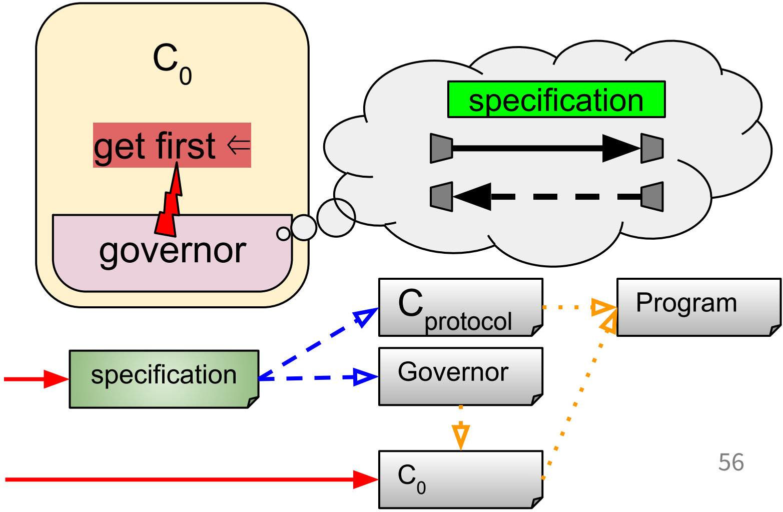
# End Slide: Summary

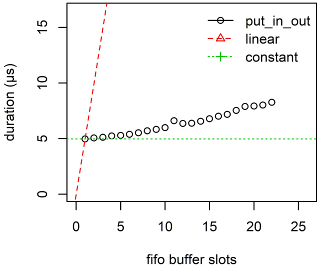
- Contributed:
  - Reo compiler **backend** to Rust.
  - Rust library for generic protocol types and behaviors (eg: `build`).
  - Performance **benchmarks** for Rust protocol objects.
  - Design of static **governors** in Rust’s type system.

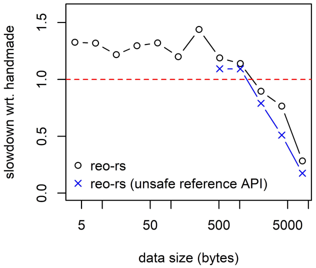


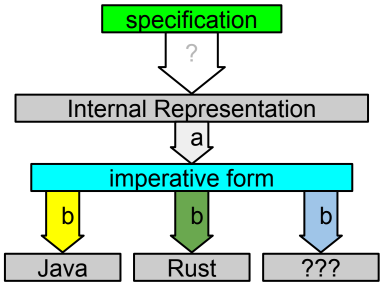
| | mean active time | | | run | mean |
|---|---|---|---|---|---|
| | p0 | p1 | p2 | duration | parallelism |
| move | 2.68μs | 2.594μs | 2.993μs | 31.1705ms | 2.652 |
| copy | 2.737μs | 2.4μs | 2.673μs | 28.7161ms | 2.720 |
| signal | 2.351μs | 2.282μs | 1.943μs | 24.7852ms | 2.653 |
| clone | 4.451ms | 4.461ms | 4.416ms | 44.609s | 2.988 |







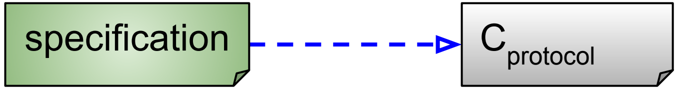




# Extra Slides

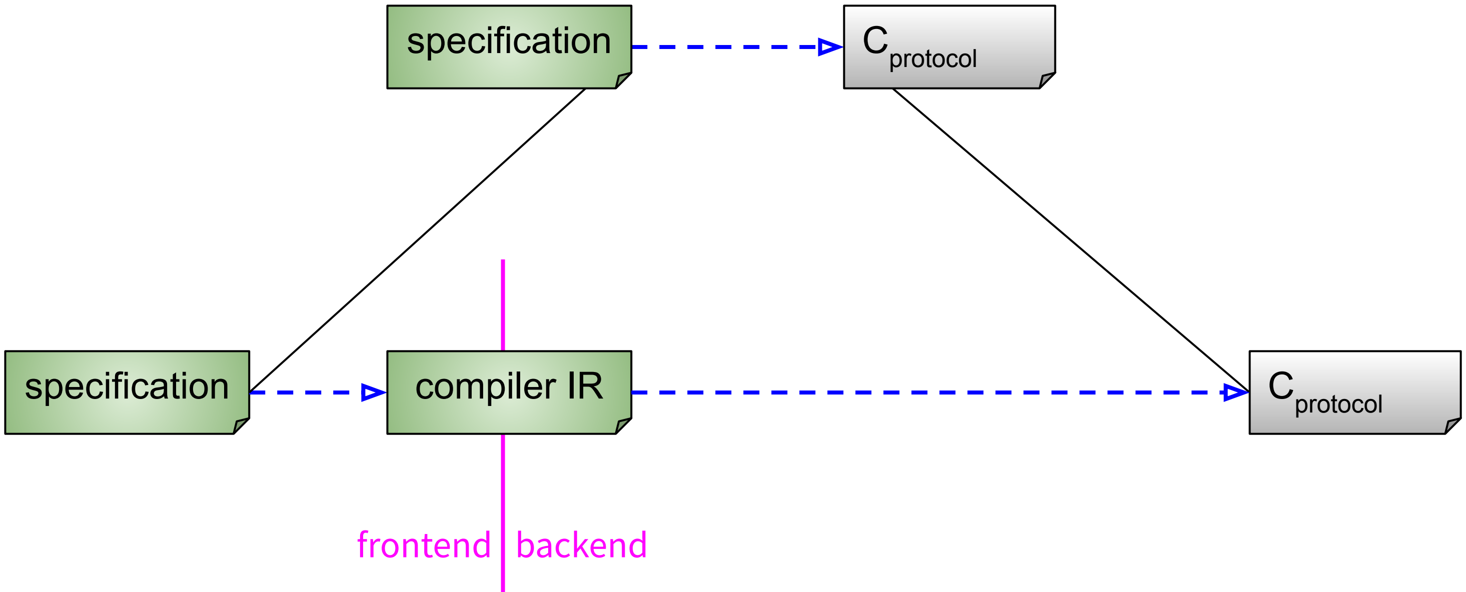
# Extra ●○ Translation Pipeline

- Translation from Reo to Rust takes several steps



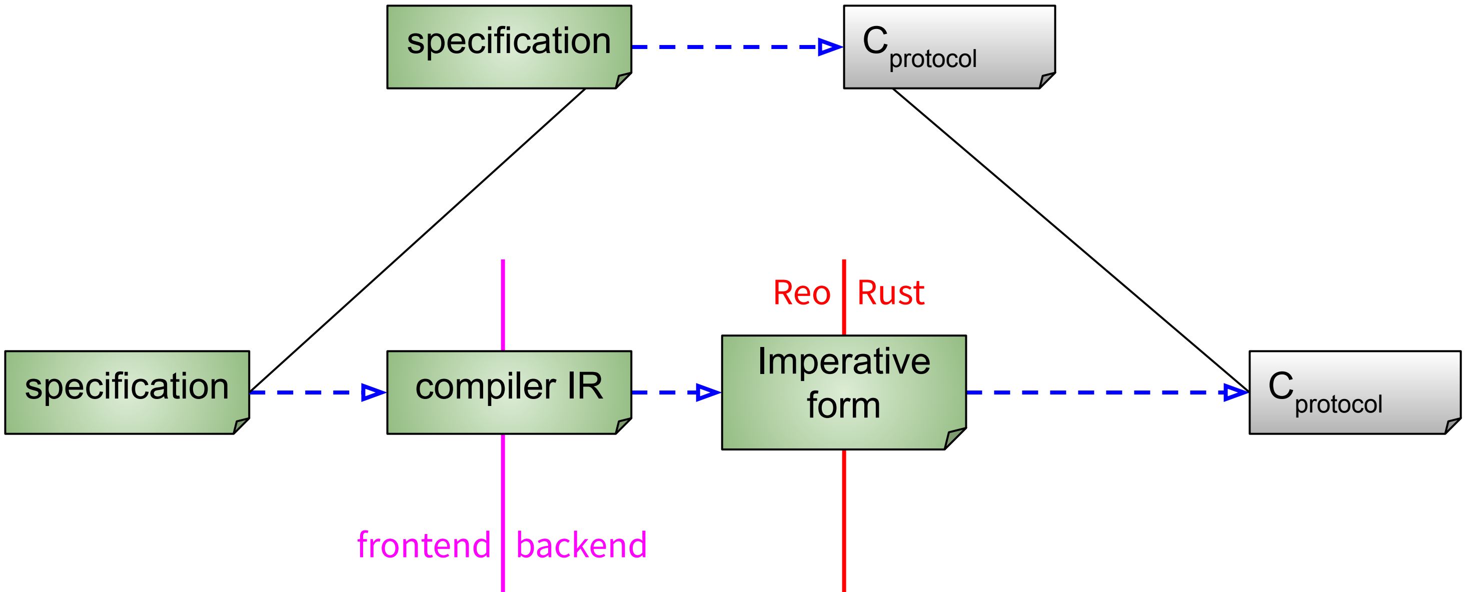
# Extra ●○ Translation Pipeline

- Translation from Reo to Rust takes several steps
  - This work takes the **frontend**'s work as given



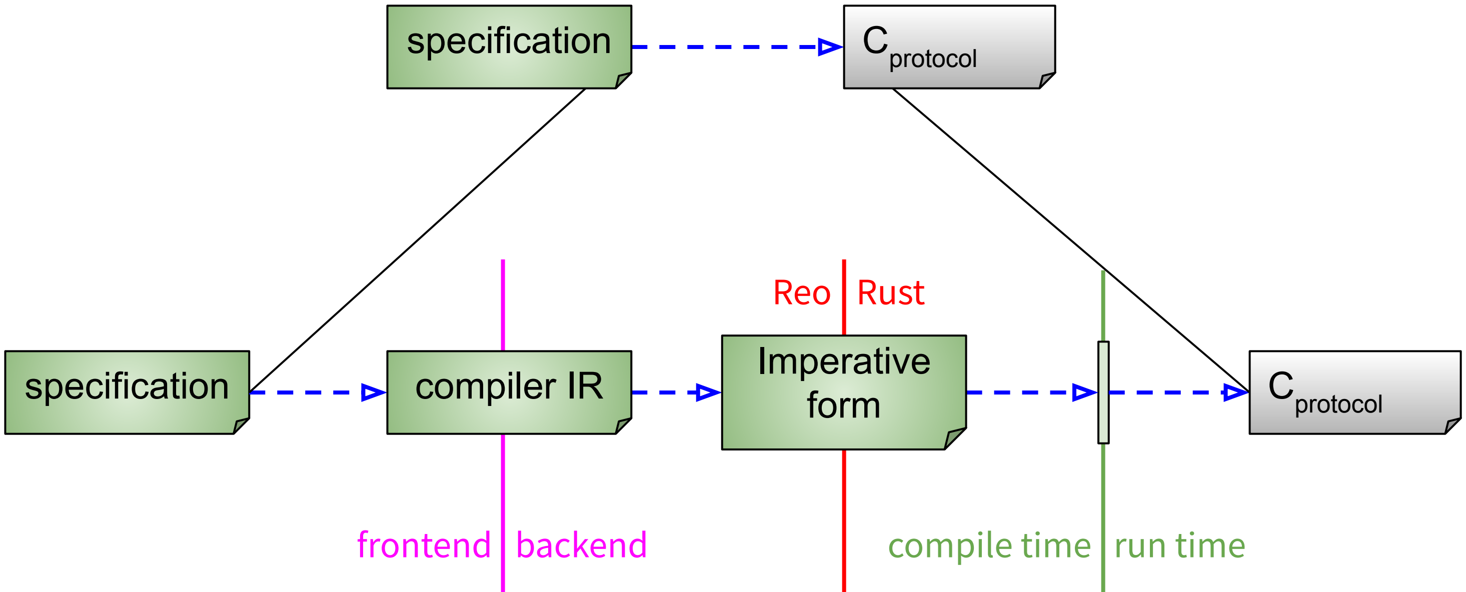
# Extra ●○ Translation Pipeline

- Translation from Reo to Rust takes several steps
  - This work takes the **frontend**’s work as given
  - Imperative form represents translation *before* rust-specifics + optimization



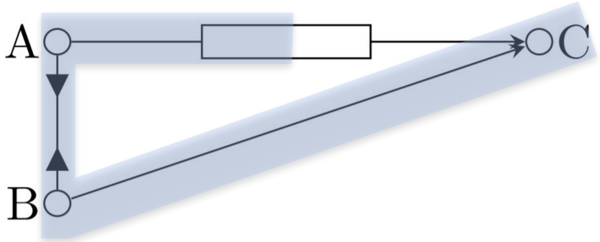
# Extra ●○ Translation Pipeline

- Translation from Reo to Rust takes several steps
    - This work takes the **frontend**’s work as given
    - Imperative form represents translation *before* rust-specifics + optimization
    - Allows us to finish translation at runtime (for example)



# Extra ●● Imperative Form

- New intermediate form for protocols
  - Between Reo specification and generated target code
  - Interactions → sequences of abstract actions



translate

**abort if** ready ⊂ {A,B,C}

**abort if** M is full

move {A → M, B → C}