Verifying Refined Multiparty Protocols Statically in F*

Fangyi Zhou
Imperial College London

Joint work with Francisco Ferreira, Raymond Hu,
Rumyana Neykova, Nobuko Yoshida

Imperial College London
University of Hertfordshire UH
Brunel University London

Let’s explain the title!

Multiparty Session Types

- Provides a global choreography for multiparty protocols
- Each participant can get a local type from the global type and be implemented independently
- Provides safety guarantees:
  - Session fidelity, deadlock freedom …

Scribble

- A description language for multiparty protocols
- Based on MPST theory, but has various extensions

Two Buyers Protocol

```
global protocol TwoBuyer (role A, role B, role S) {
    BookId     (id: int) from A to S;
    QuoteA     (x : int) from S to A;
    QuoteB     (y : int) from S to B;
    ProposeA   (a : int) from A to B;
    choice at B {
        Ok     (b : int) from B to A;
        Buy    ()        from A to S;
    } or {
        No     ()        from B to A;
        Cancel ()        from A to S;
    }
}
```

Refinement Types

- Build upon an existing type system
- Allow base types to be refined via predicates
  - e.g. Integers can be refined to even and odd
- Specify data dependencies
- Example: Liquid Haskell [Vazou et al. 2014]

[Vazou et al. 2014]: Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell.

A taste of refinement types

$$\{\nu : b \mid M\}$$

Base type b, value ν refined by term M

- The integer literal `1`
  - A possible type: $\{\nu : \textbf{int} \mid \nu = 1\}$
  - Another possible type: $\{\nu : \textbf{int} \mid \nu \geq 0\}$
  - Or more… $\{\nu : \textbf{int} \mid \textbf{true}\}$

Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. 2012.
Semantic subtyping with an SMT solver

Why do we want refined protocols?

Make more precise protocol specification

Two Buyers Protocol

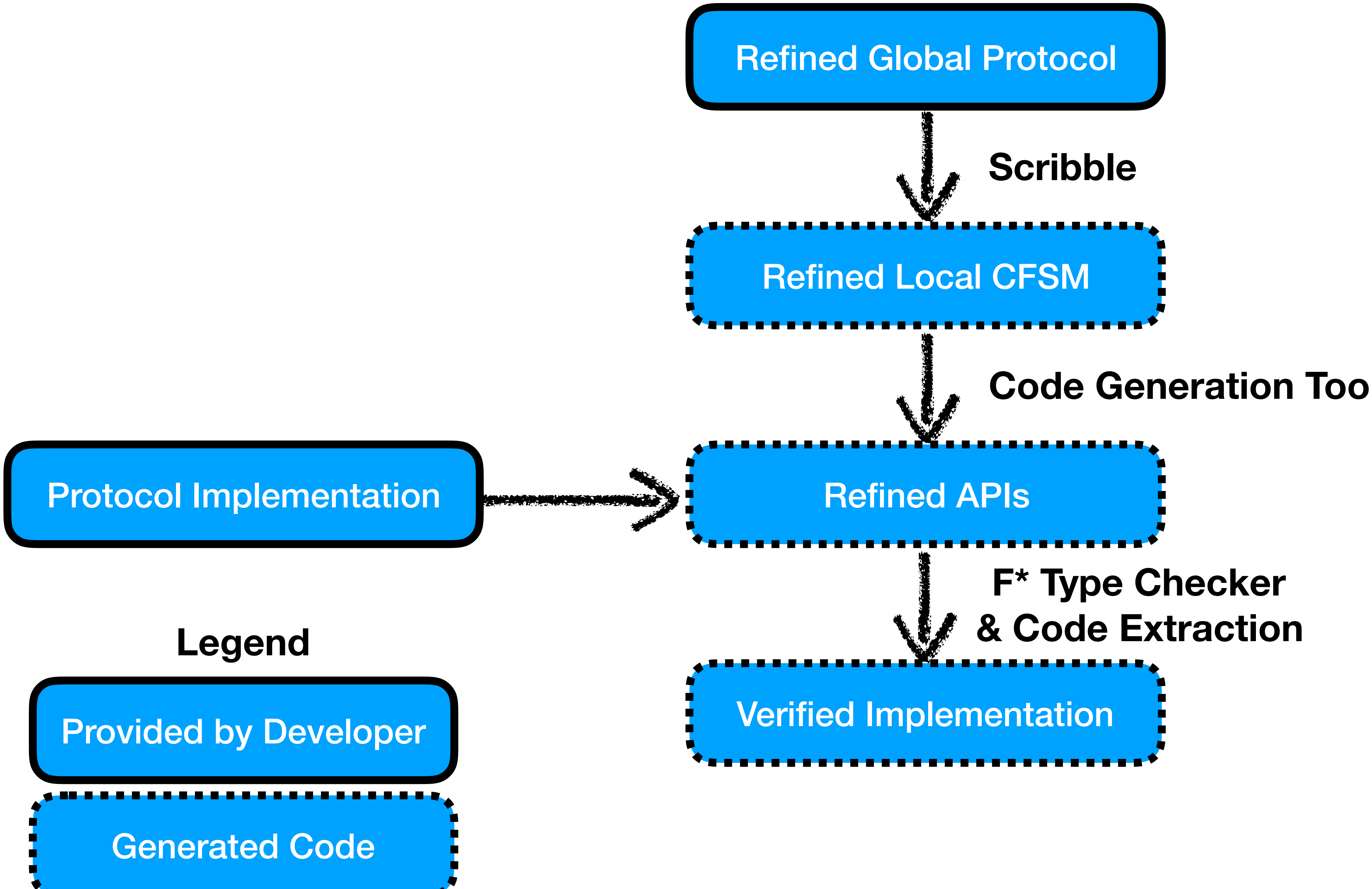
```
global protocol TwoBuyer (role A, role B, role S) {
    BookId     (id: int) from A to S;
    QuoteA     (x : int) from S to A; @"x ⩾ 0"
    QuoteB     (y : int) from S to B; @"x=y"
    ProposeA   (a : int) from A to B; @"a ⩾ 0 && a ⩽ x"
    choice at B {
        Ok     (b : int) from B to A; @"b=y-a && y-a ⩽ a"
        Buy    ()        from A to S;
    } or {
        No     ()        from B to A;
        Cancel ()        from A to S;
    }
}
```

Refinements

What is F*, and why use F*?

- F* is a programming language designed for verification
- SMT-based verification
- Support for refinement types and effectful programs
- Extraction to OCaml

Workflow


Refined Global Protocol
Scribble
Refined Local CFSM
Code Generation Tool
Protocol Implementation
Refined APIs
F* Type Checker
& Code Extraction
Verified Implementation
Legend
Provided by Developer
Generated Code

Two Buyers Protocol

```
global protocol TwoBuyer (role A, role B, role S) {
    BookId     (id: int) from A to S;
    QuoteA     (x: int)  from S to A; @"x≥0"
    QuoteB     (y: int)  from S to B; @"x==y"
    ProposeA   (a: int)  from A to B; @"a≥0 && a≤x"
    choice at B {
        Ok     (b: int)  from B to A; @"b==y-a && y-a≤a"
        Buy    ()        from A to S;
    } or {
        No     ()        from B to A;
        Cancel ()        from A to S;
    }
}
```

Projection on B

```
global protocol TwoBuyer (role A, role B, role S) {
    BookId     (id: int) from A to S;
    QuoteA     (x: int)  from S to A; @"x ⩾ 0"
    QuoteB     (y: int)  from S to B; @"x==y"
    ProposeA   (a: int)  from A to B; @"a ⩾ 0 && a ⩽ x"
    choice at B {
        Ok     (b: int)  from B to A; @"b==y-a && y-a ⩽ a"
        Buy    ()        from A to S;
    } or {
        No     ()        from B to A;
        Cancel ()        from A to S;
    }
}
```

Communicating Finite State Machine (CFSM)

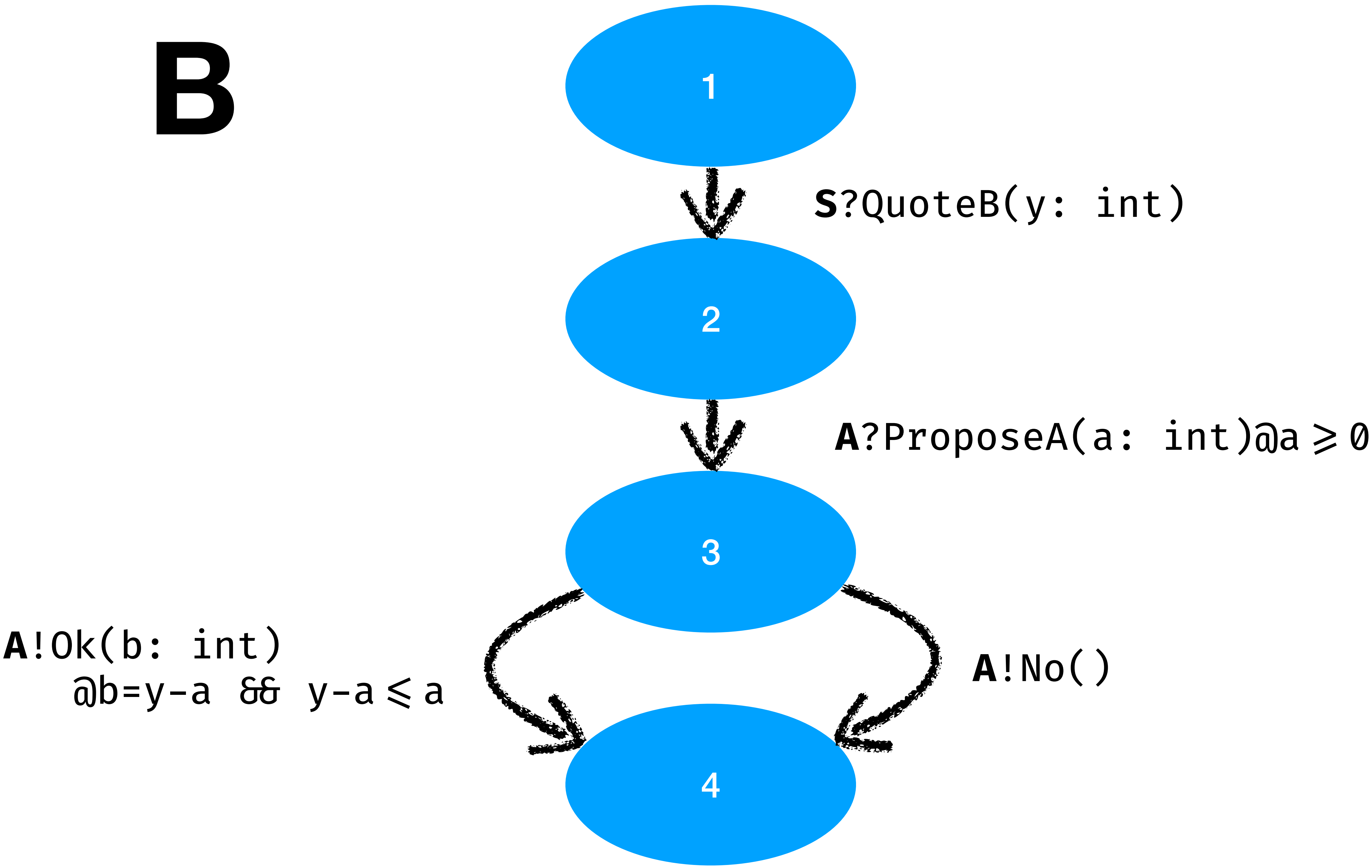
- Local types have correspondent basic CFSMs [Deniélou and Yoshida 2013]
- Transitions are sending/receiving actions
- CFSM-based code generation approach is a common technique (e.g. [Hu and Yoshida 2016])

[Deniélou and Yoshida 2013]: Pierre-Malo Deniélou, and Nobuko Yoshida. 2013.
Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types.

[Hu and Yoshida 2016]: Raymond Hu, and Nobuko Yoshida. 2016.
Hybrid Session Verification Through Endpoint API Generation.

CFSM Representation

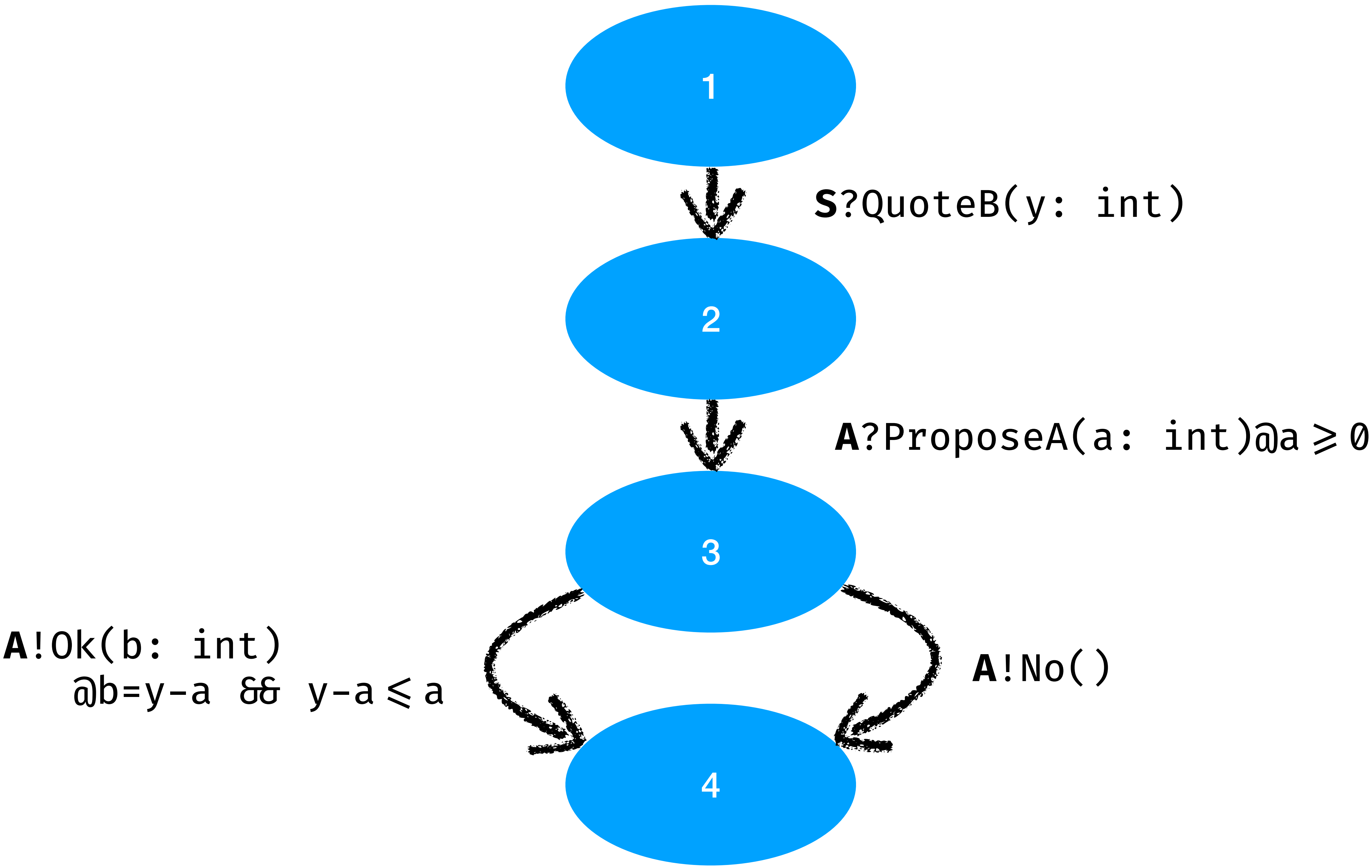
B



Event based APIs from CFSM

- States:
  - Records variables from previous communications
- Transitions:
  - Provide callbacks for handing state transitions
- Assertions are converted to refinement types
- No exposed states (i.e. linear by construction)

CFSM Representation



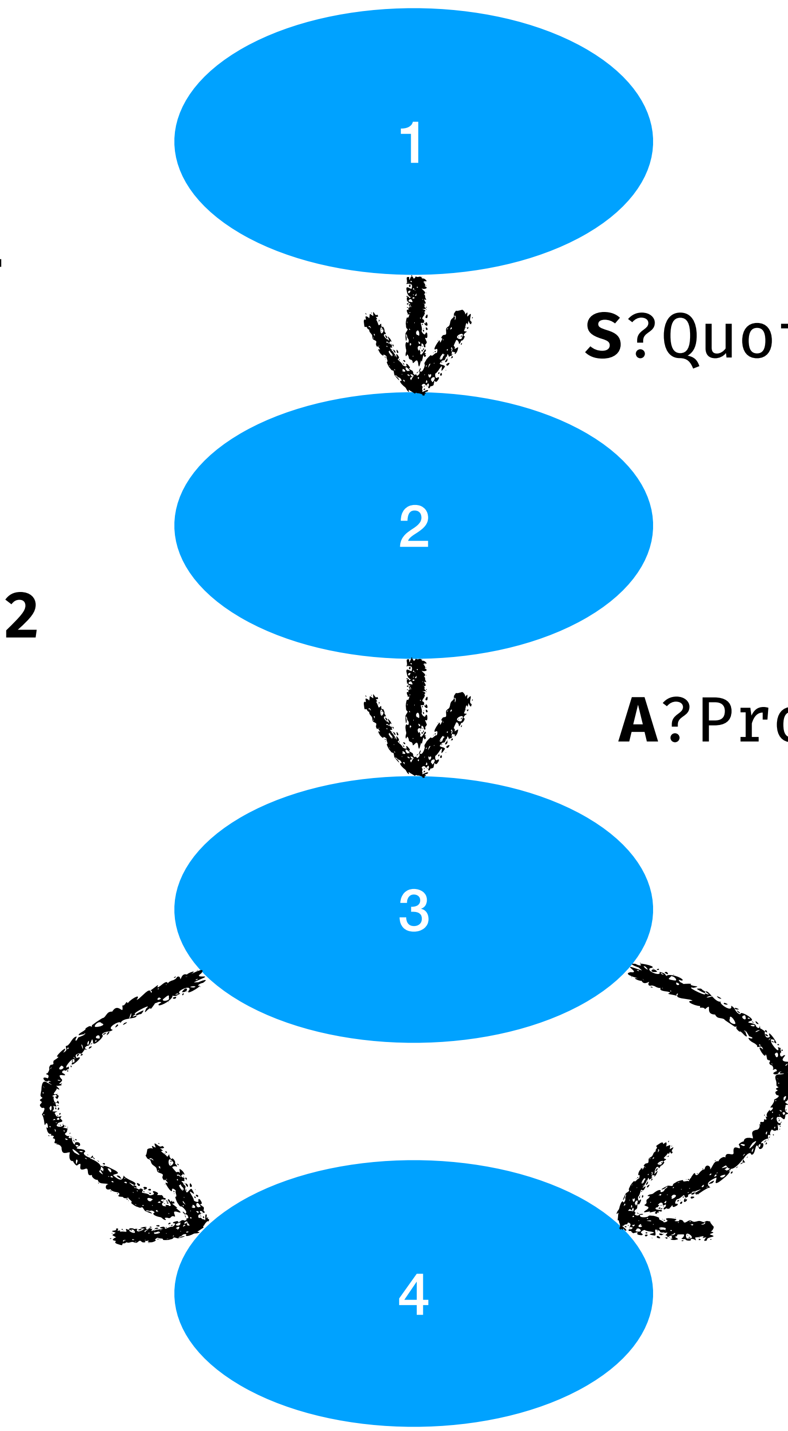
Event based APIs from CFSM

- States:
  - Records variables from previous communications
- Transitions:
  - **Provide callbacks for handing state transitions**
- Assertions are converted to refinement types
- No exposed states (i.e. linear by construction)

API Generation

```
receiveQuoteB: State1
            → int
            → unit
```

```
receiveProposeA: State2
               → int
               → unit
```



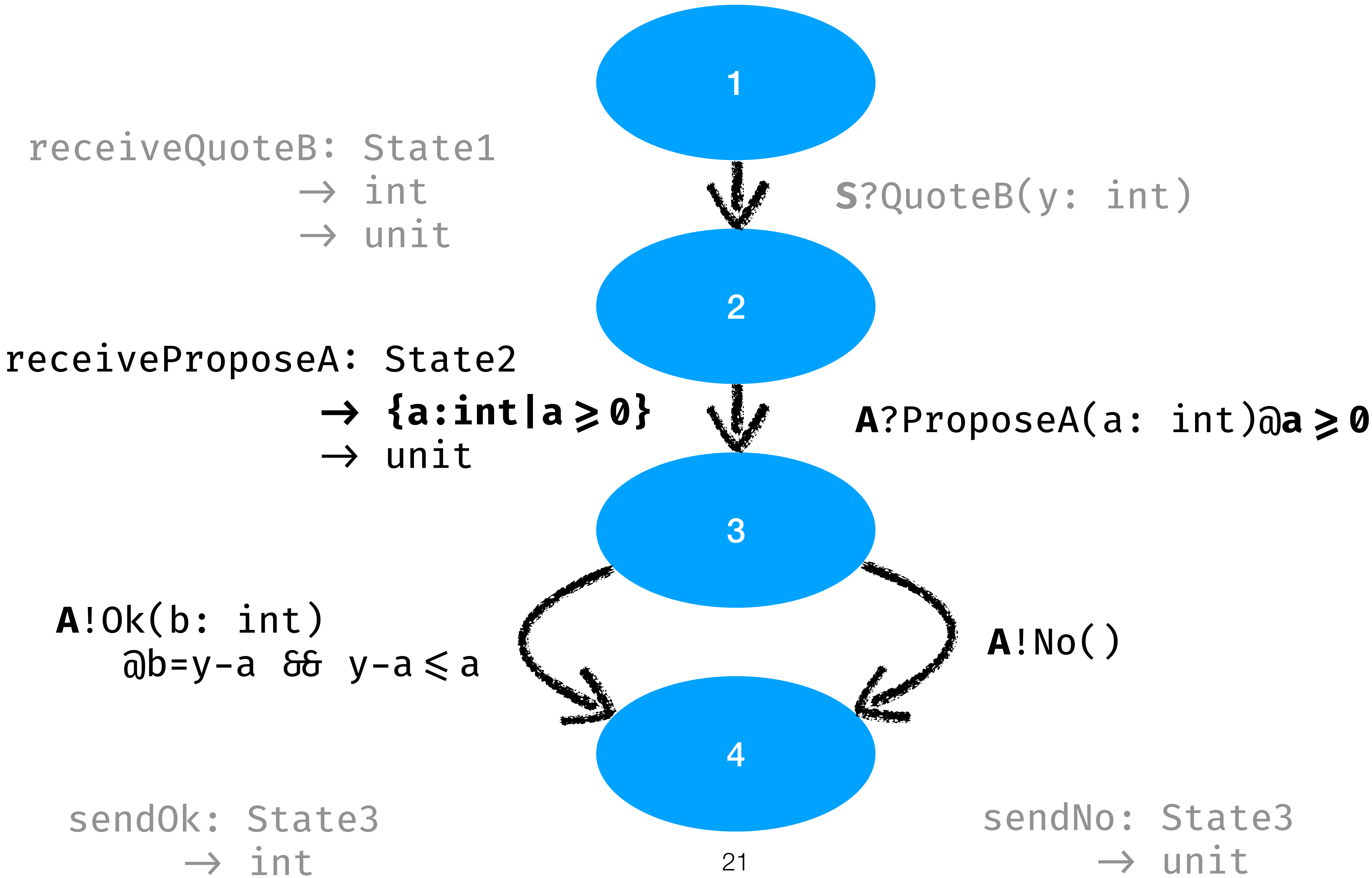

```
sendOk: State3
      → int
```

```
sendNo: State3
      → unit
```

Event based APIs from CFSM

- States:
  - Records variables from previous communications
- Transitions:
  - Provide callbacks for handing state transitions
- **Assertions are converted to refinement types**
- No exposed states (i.e. linear by construction)

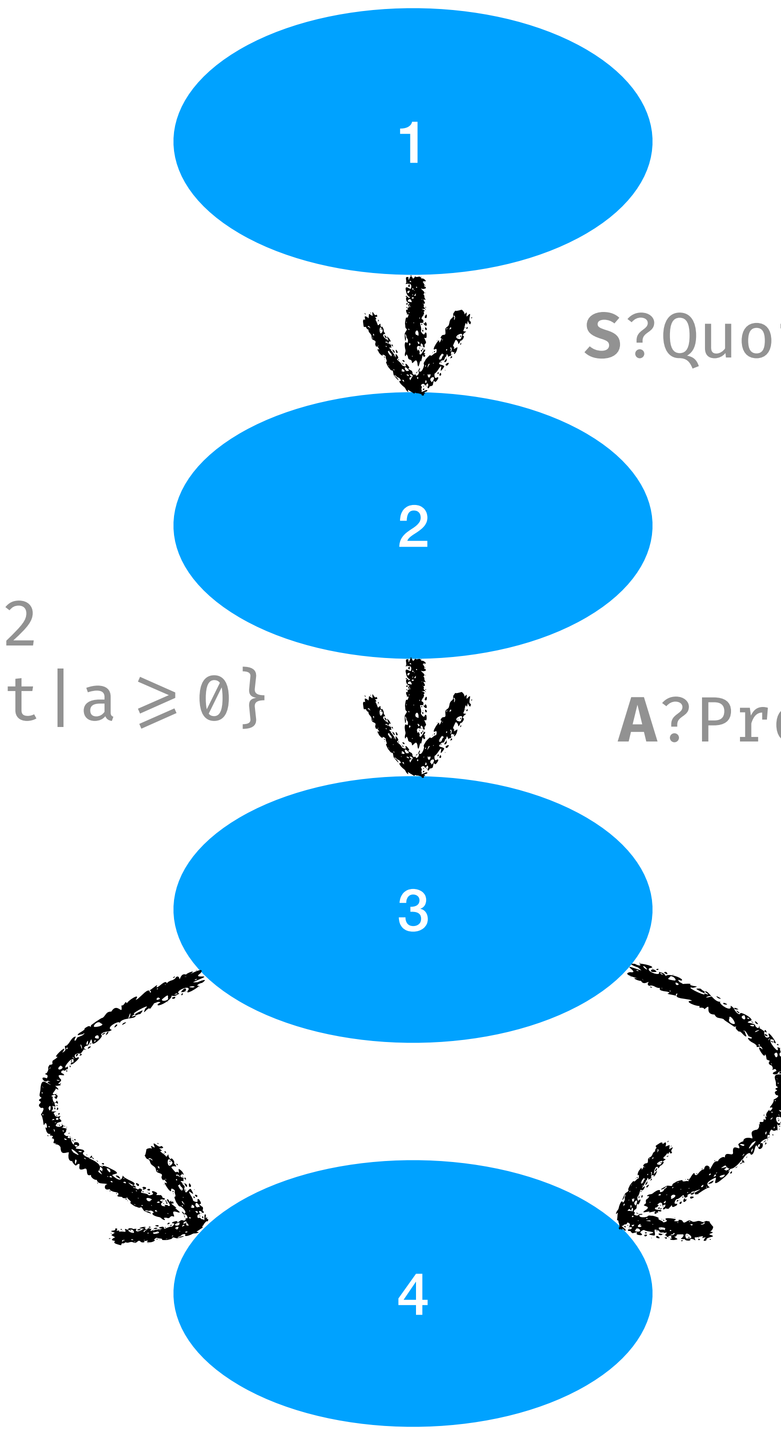
API Generation



Event based APIs from CFSM

- States:
  - **Records variables from previous communications**
- Transitions:
  - Provide callbacks for handing state transitions
- Assertions are converted to refinement types
- No exposed states (i.e. linear by construction)

API Generation




```
receiveQuoteB: State1
             → int
             → unit
```

S?QuoteB(y: int)

```
receiveProposeA: State2
               → {a:int|a ⩾ 0}
               → unit
```

A?ProposeA(a: int)@a ⩾ 0

A!Ok(b: int)
@b=y-a && y-a ⩽ a

sendOk: (st: State3)
→ {b:int|b=st.y-st.a && st.y-st.a ⩽ st.a}

A!No()

```
sendNo: State3
      → unit
```

Generated APIs

```
type Handlers = {

   state1OnReceiveQuoteB : State1 → int → unit

   [<Refined("State2 → (a: {a:int | a ≥ 0}) → unit")>]
   state2OnReceiveProposeA : State2 → int → unit

   [<Refined("(state: State3)
           → {label:int | (label==Ok && st.y-st.a ≤ st.a)
                          || (label==No)}”)>]
   state3 : State3 → State3Choice

   [<Refined("(st: State3Ok)
           → {b:int | b==st.y-st.a && b ≤ st.a}”)>]
   state3OnSendOk : State3Ok → int

   state3OnSendNo : State3No → unit

}
```

Ongoing Work

- **Improving Expressiveness - Invariants on Recursive Protocols:**
    - Protocols have state variables and protocol invariants
    - State variables can carry to subsequent iterations

```
global protocol Fib(role A, role B) @"<x:=0, y:=1> x≥0 && y≥x"
{
  1(x1: int) from A to B; @"x1==x"
  2(y1: int) from A to B; @"y1==y"
  3(z1: int) from B to A; @"z1==x1+y1"
  do Fib(A, B);           @"<y, z1>"
}
```

Ongoing Work

- Formalisation of refined multiparty session types
  - Syntax and LTS semantics of global and local types
  - Trace equivalence proofs of global/local type semantics

Related Work

- Assertion-based Calculus [Bocchi et al. 2010]
    - A theory of multiparty session calculus with assertions
- Session Type Provider [Neykova et al. 2018]
    - Refinements checked dynamically during execution

[Bocchi et al. 2010]: Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010.
A Theory of Design-by-Contract for Distributed Multiparty Interactions
[Neykova et al. 2018]: Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018.
A session type provider: compile-time API generation of distributed protocols with refinements in F#

Conclusion

- We presented a toolchain that
  - allows refinement types in global protocols
  - checks refinement statically with F*
  - guarantees linearity of channel usage by construction

Thank you!