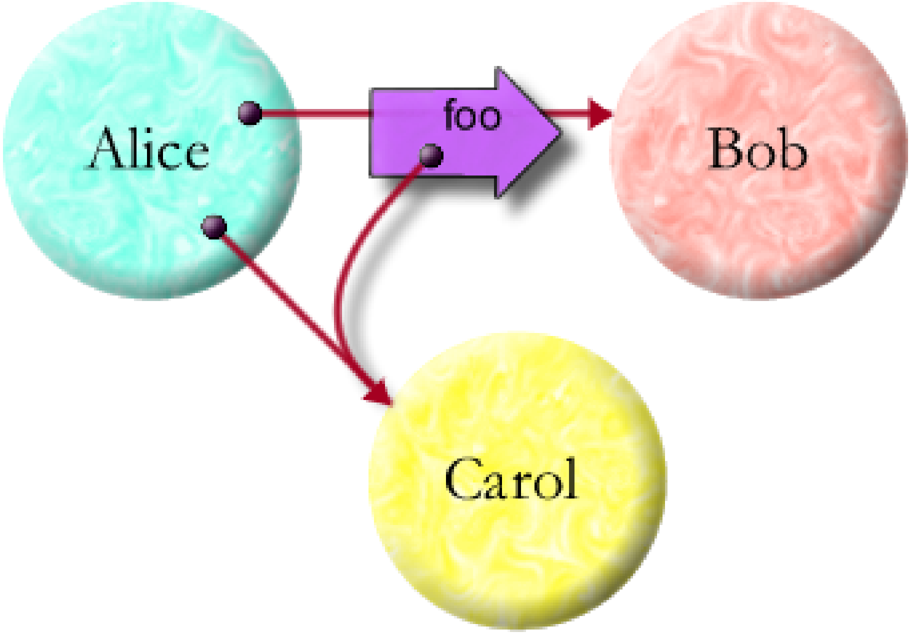
Robust Composition:

Towards a Unified Approach to Access Control and Concurrency Control

by Mark S. Miller



Tuesday March 28, 2006

Talk Overview

- Research question
  - Programs as plans. Plan interference hazards
  - Controlling access & concurrency
- Approach
  - Robust composition by controlling interaction
- Example & Demo (time permitting)

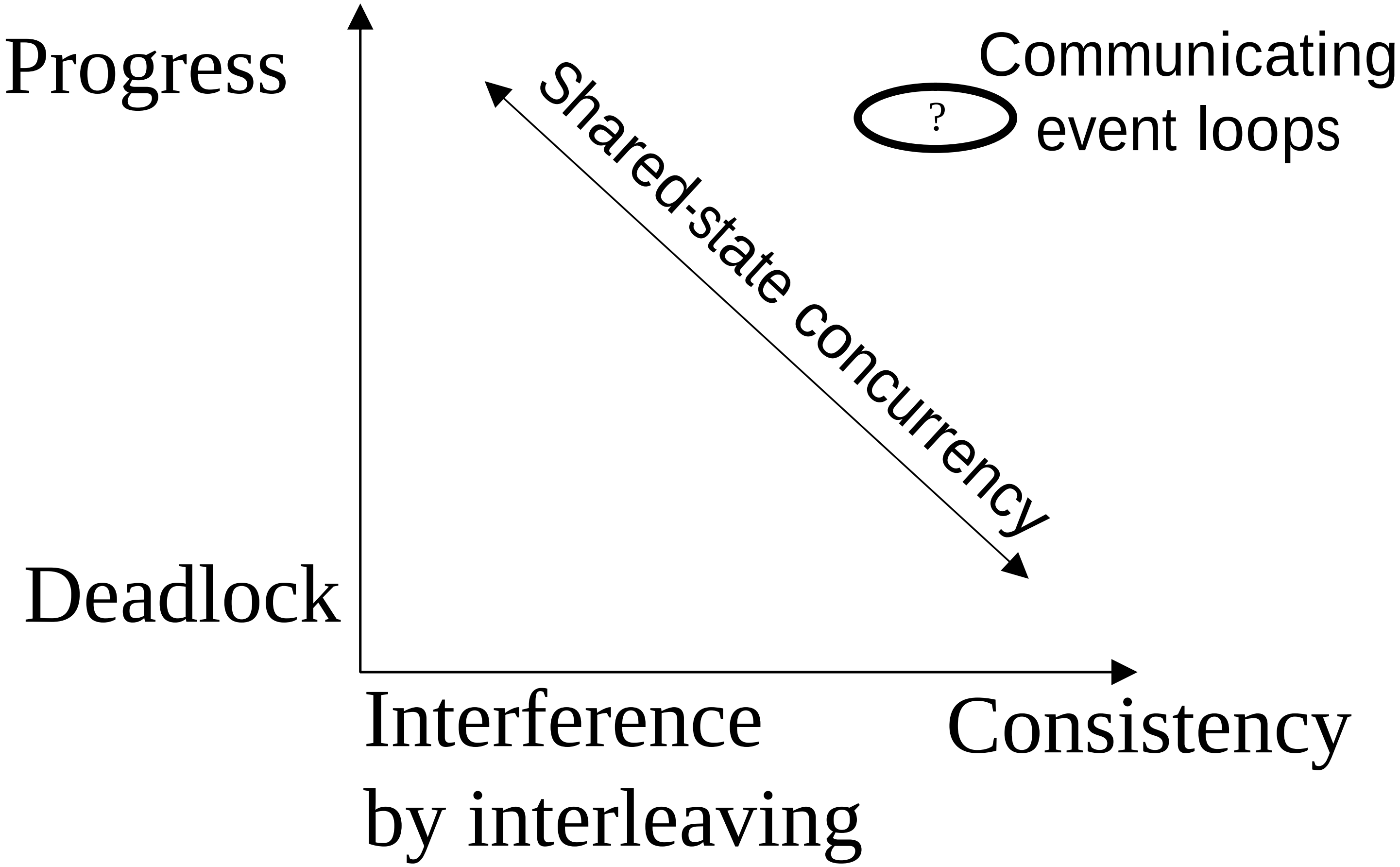
Research Question

- Programmers express plans for machines to run
    - Plan must handle all relevant contingencies
    - Danger: explosive case analysis
- Plan Coordination =

    plan composition (realize cooperative opportunities)

    + plan separation (avoid destructive interference)
- OO works “in the small” – local, sequential, benign
    - Abstraction reduces relevant cases
- Can we support plan coordination at Internet scale?
    - asynchronous, distributed, possibly malicious
    - Existing separate solutions don’t compose well

Progress vs. Consistency?
(concurrency control)



Progress

Deadlock

Shared-state concurrency

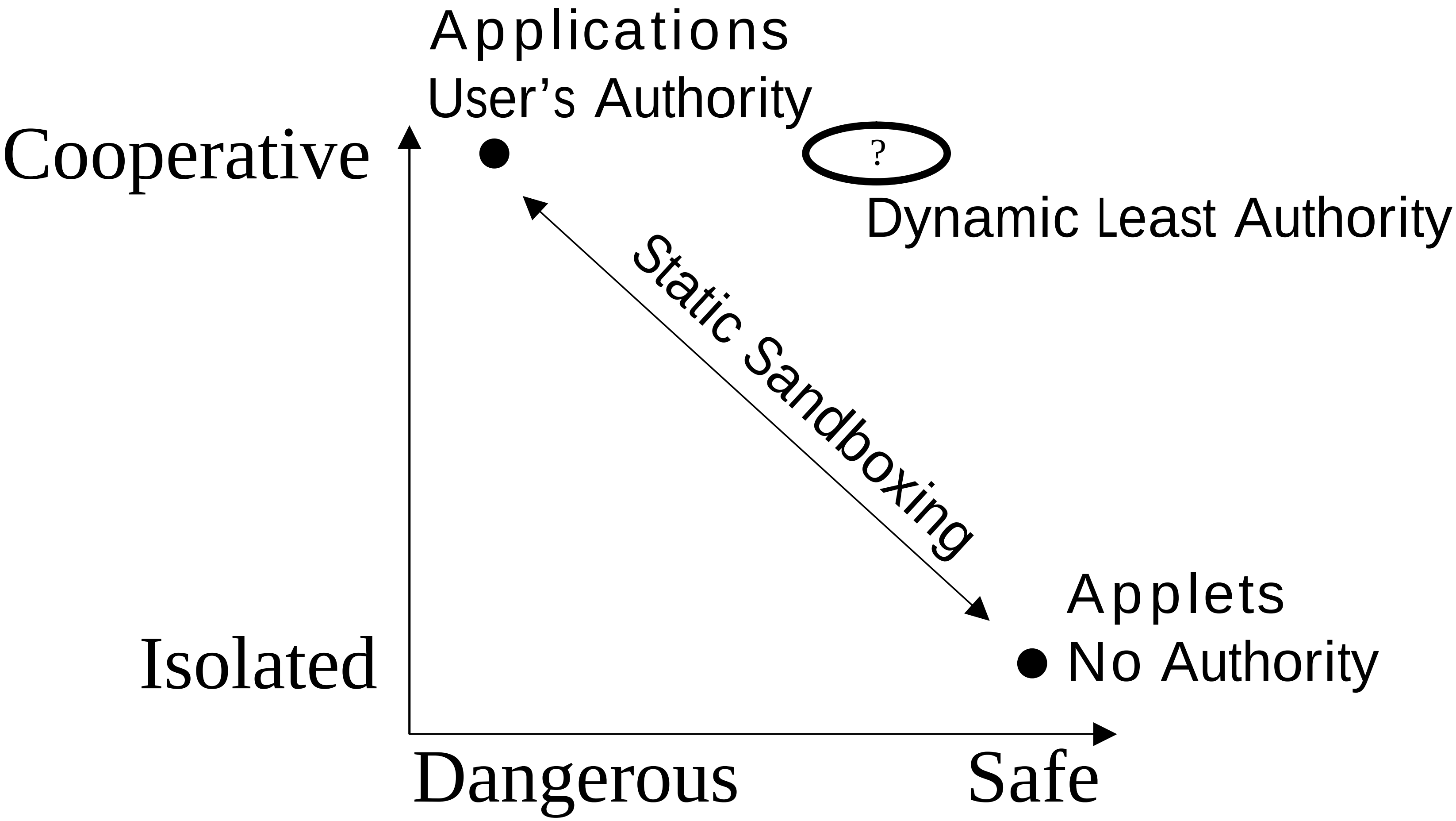
?

Communicating event loops

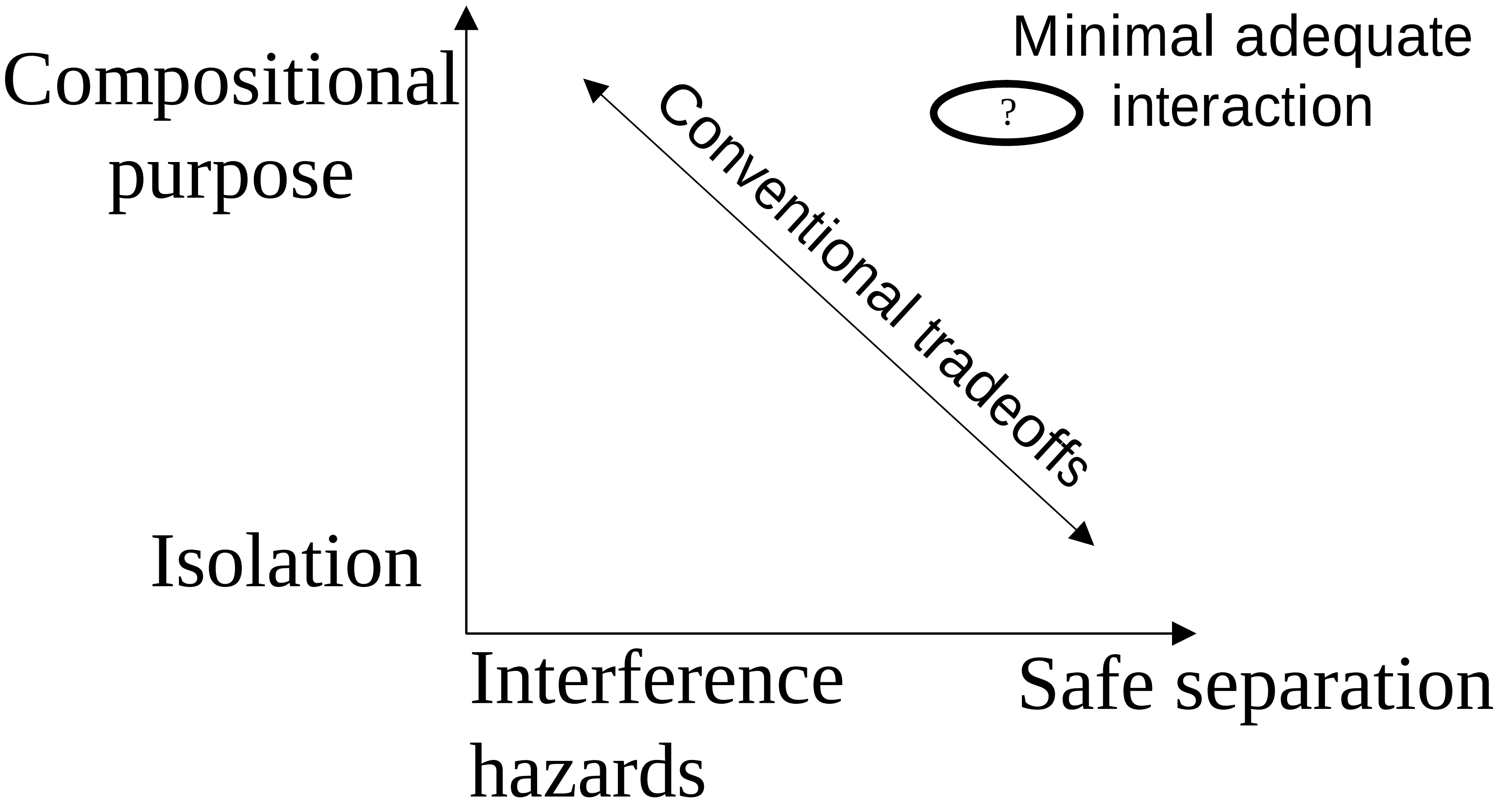
Interference by interleaving

Consistency

Functionality vs. Security?
(access control)



Purposes and Hazard (interaction control)



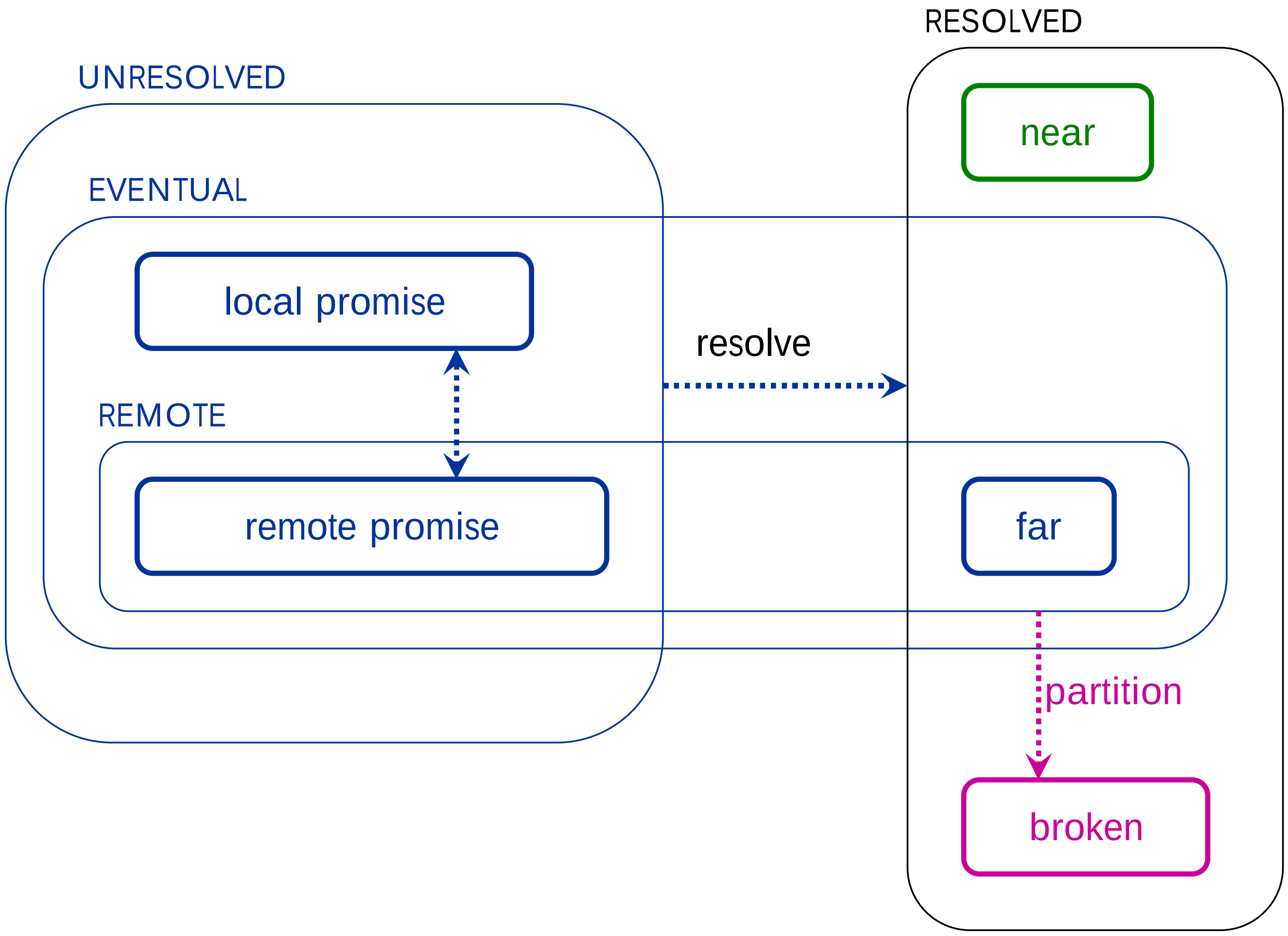
Simultaneous Problems

- Excess authority invites abuse (viruses, spyware)
- Interleaving causes inconsistency
- Excluding interleavings cause deadlock
- Inter-machine latency delays distributed plans
- Partial failures (disconnects and crashes) demand diverse recovery strategies

Novelty: Integration and linguistic support,

Interaction control by reference states & transitions

Reference states & transitions
Causal transmission depends on state



Robust Composition Challenges
for Internet-scale distributed computing

Extend virtues of oo-languages …

… among mutually suspicious objects

… on mutually suspicious machines

… without undue vulnerability.

Let objects interact asynchronously…

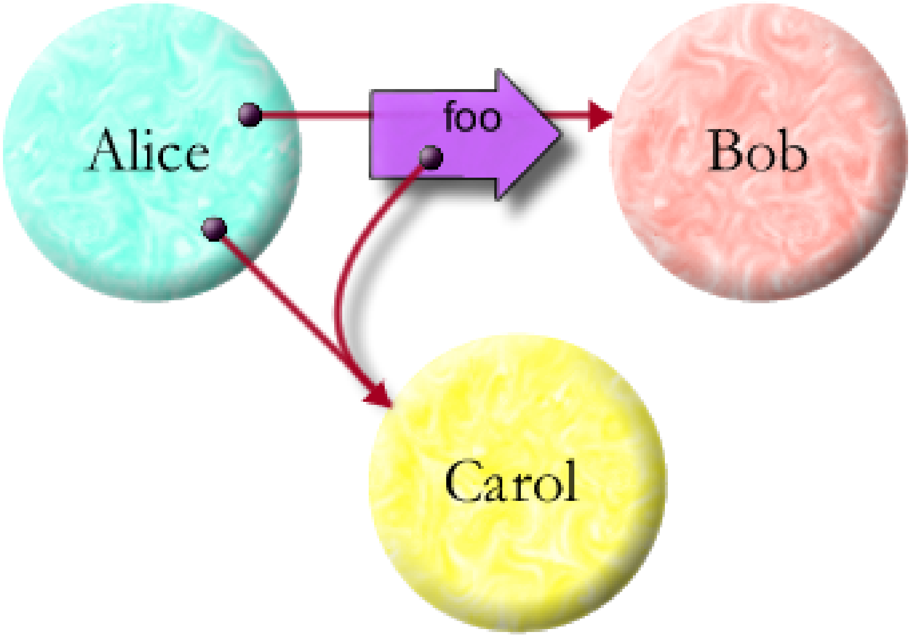
… in partially predictable order

… with distant machines

… that may not be reachable.

Extend virtues of oo-languages …

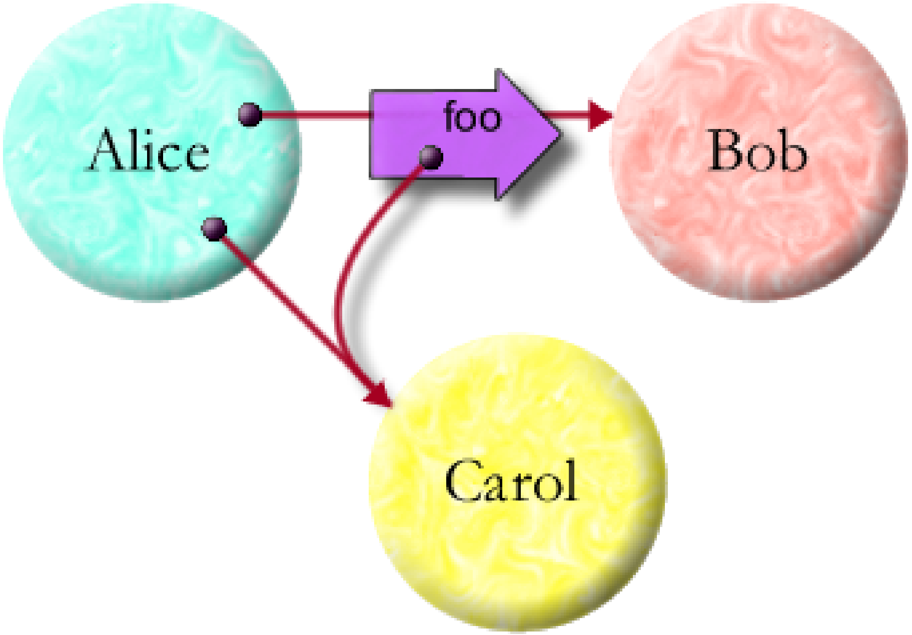
Alice says: `bob.foo(carol)`



- abstraction & composition, rapid prototyping
- precise semantics, compact familiar notation

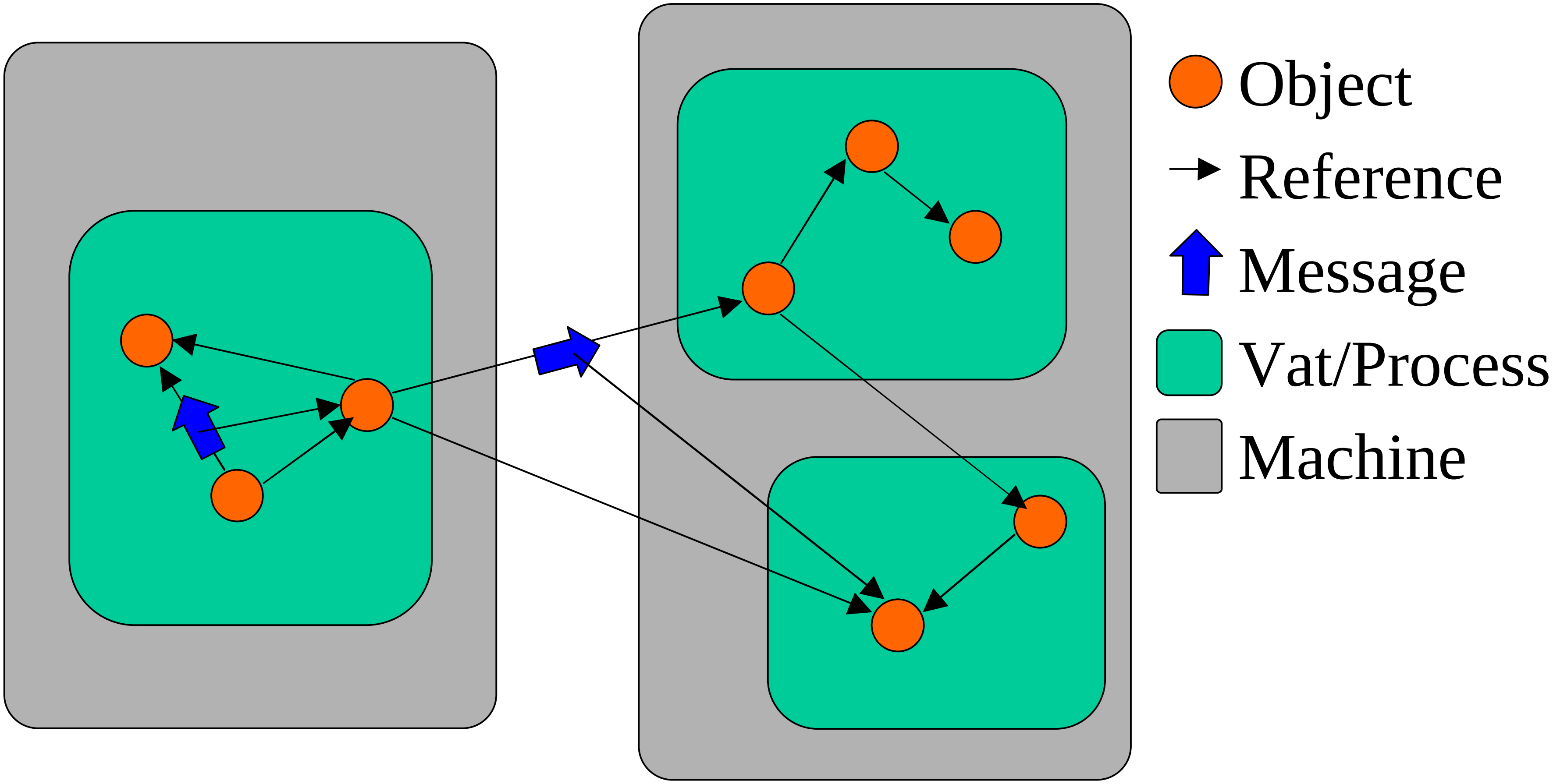
… among mutually suspicious objects …

Object-capabilities: Reference Graph == Access Graph



- Absolute encapsulation—causality only by messages
- Only references permit causality
- Graph limits what's possible

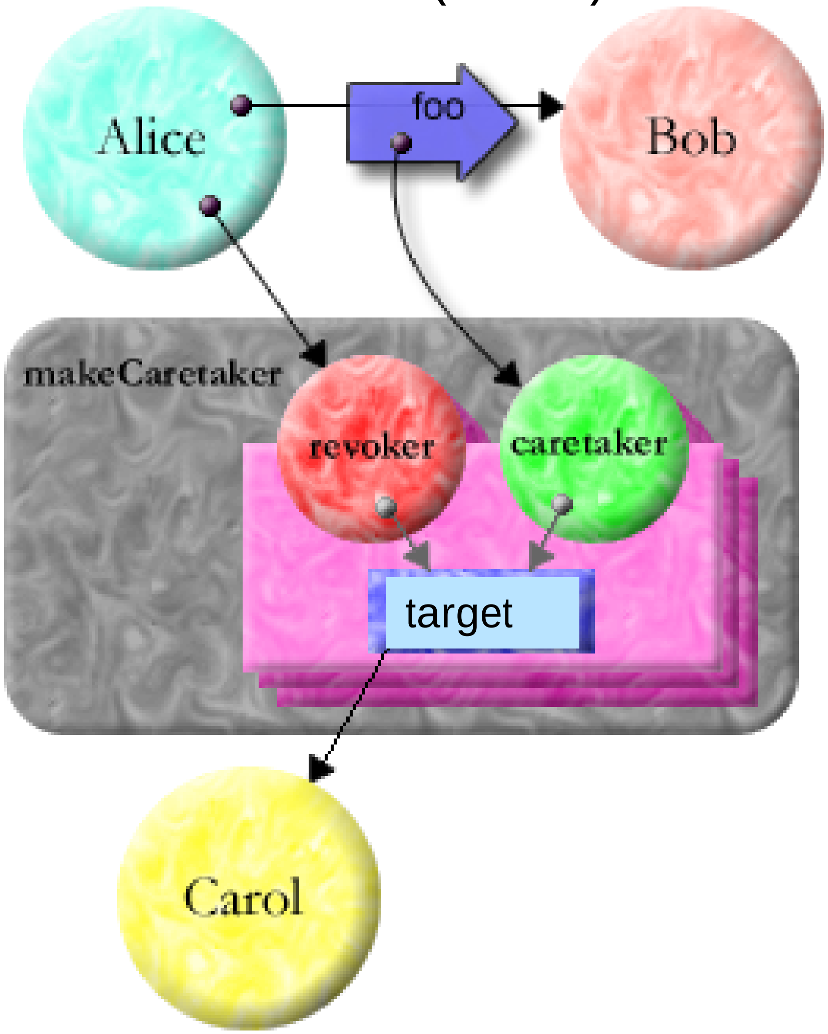
… on mutually suspicious machines …



Pluribus: cryptographic capability protocol
Kernel-E: safe mobile code

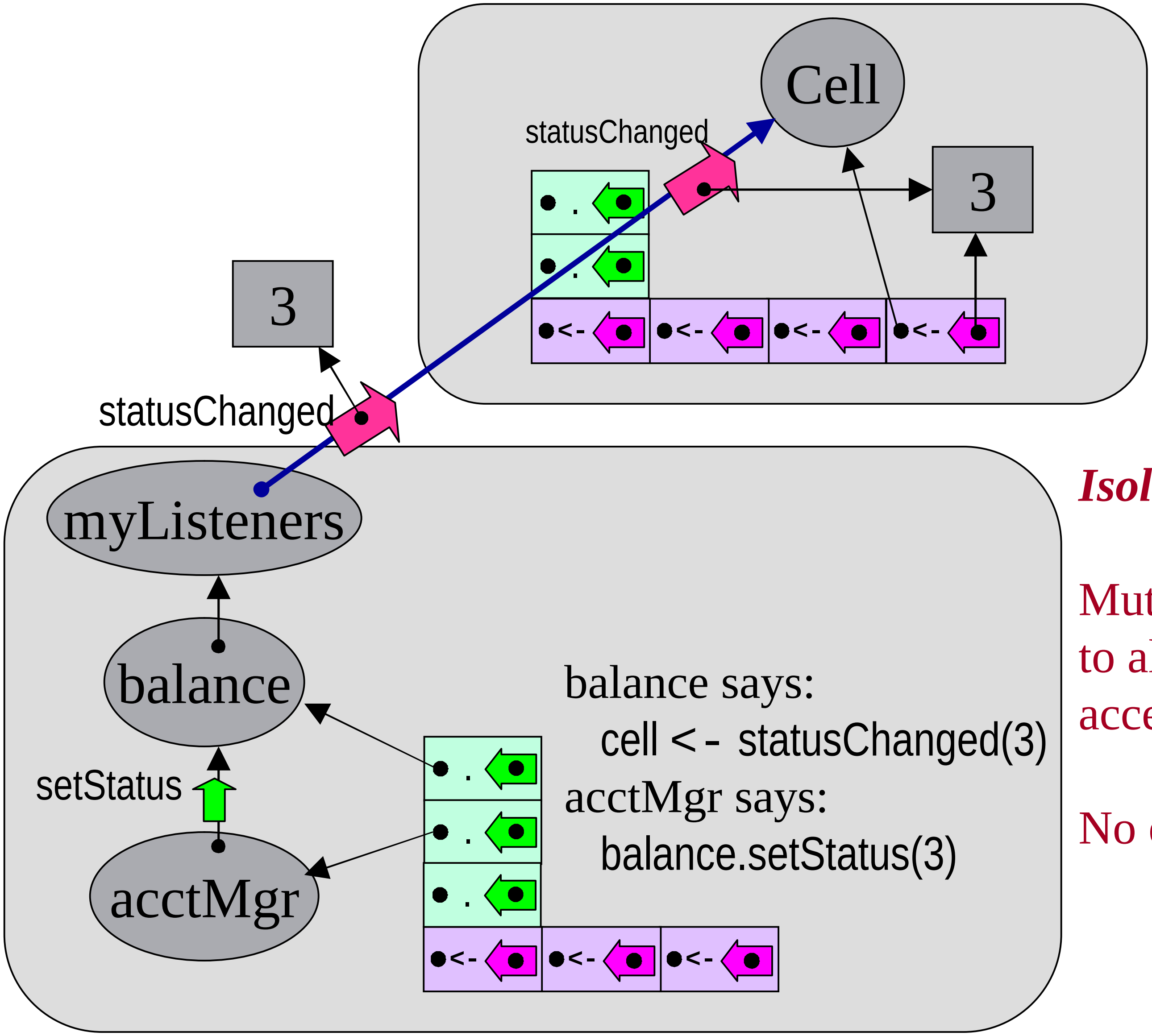
… without undue vulnerability.

Alice says: def [carol2, carol2revoker] := makeCaretaker(carol)
bob.foo(carol2)



```
def makeCaretaker(var target) :any {
    def caretaker {
        match [verb :String, args :any[]] {
            E.call(target, verb, args)
    }  }
    def revoker {
        to revoke() :void {
            target := null
    }  }
    return [caretaker, revoker]
}
```

Let objects interact asynchronously …



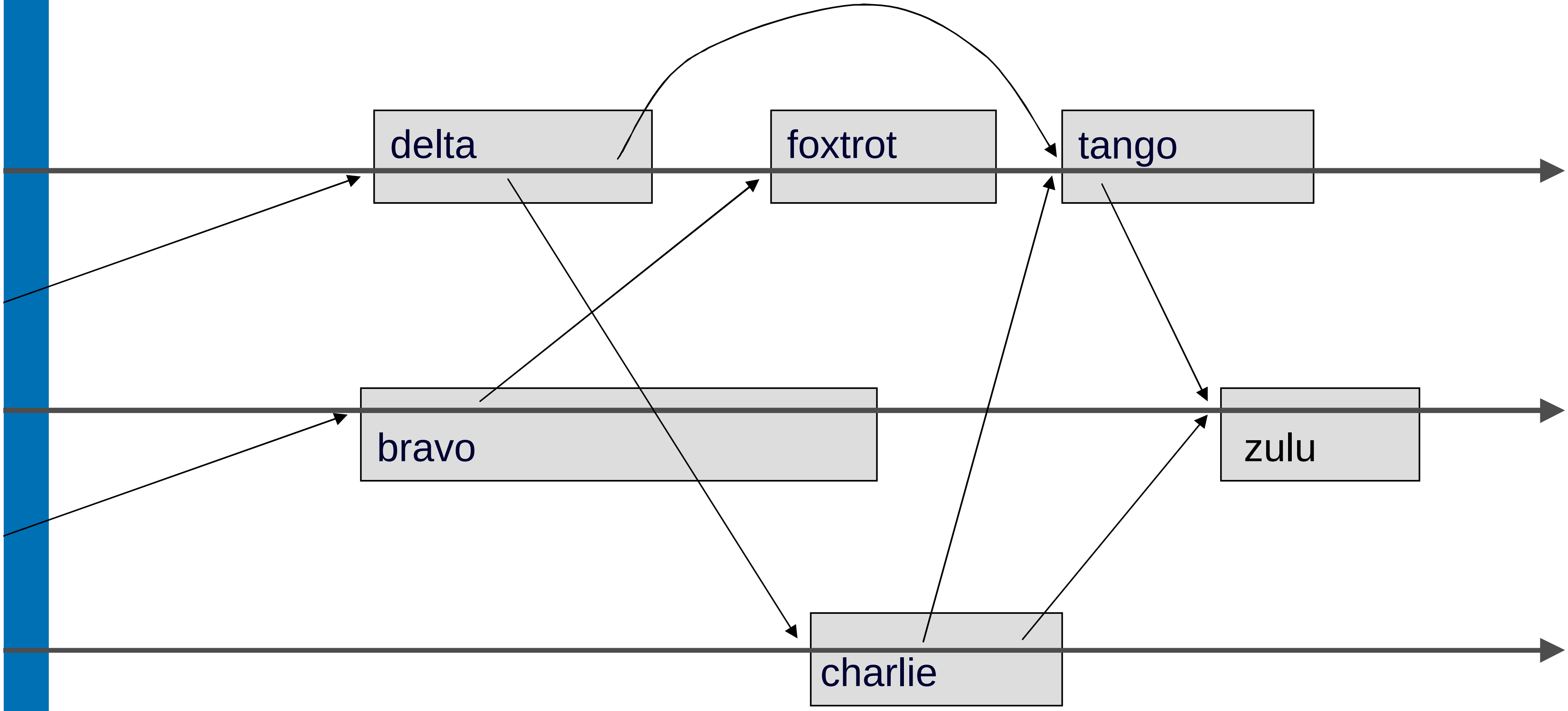
balance says:
  cell <- statusChanged(3)
acctMgr says:
  balance.setStatus(3)

Isolated Turns

Mutually exclusive access to all that's synchronously accessible

No explicit locking

Let objects interact asynchronously …

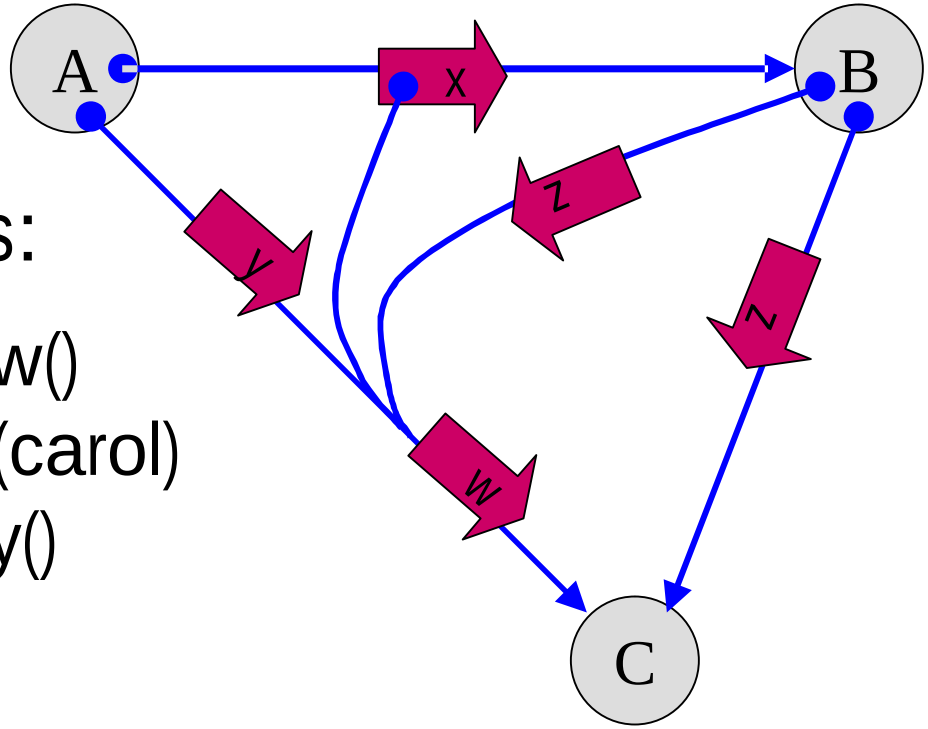


Turns are isolated units of operation

… in partially predictable order …

Alice says:

carol <- w()
bob <- x(carol)
carol <- y()

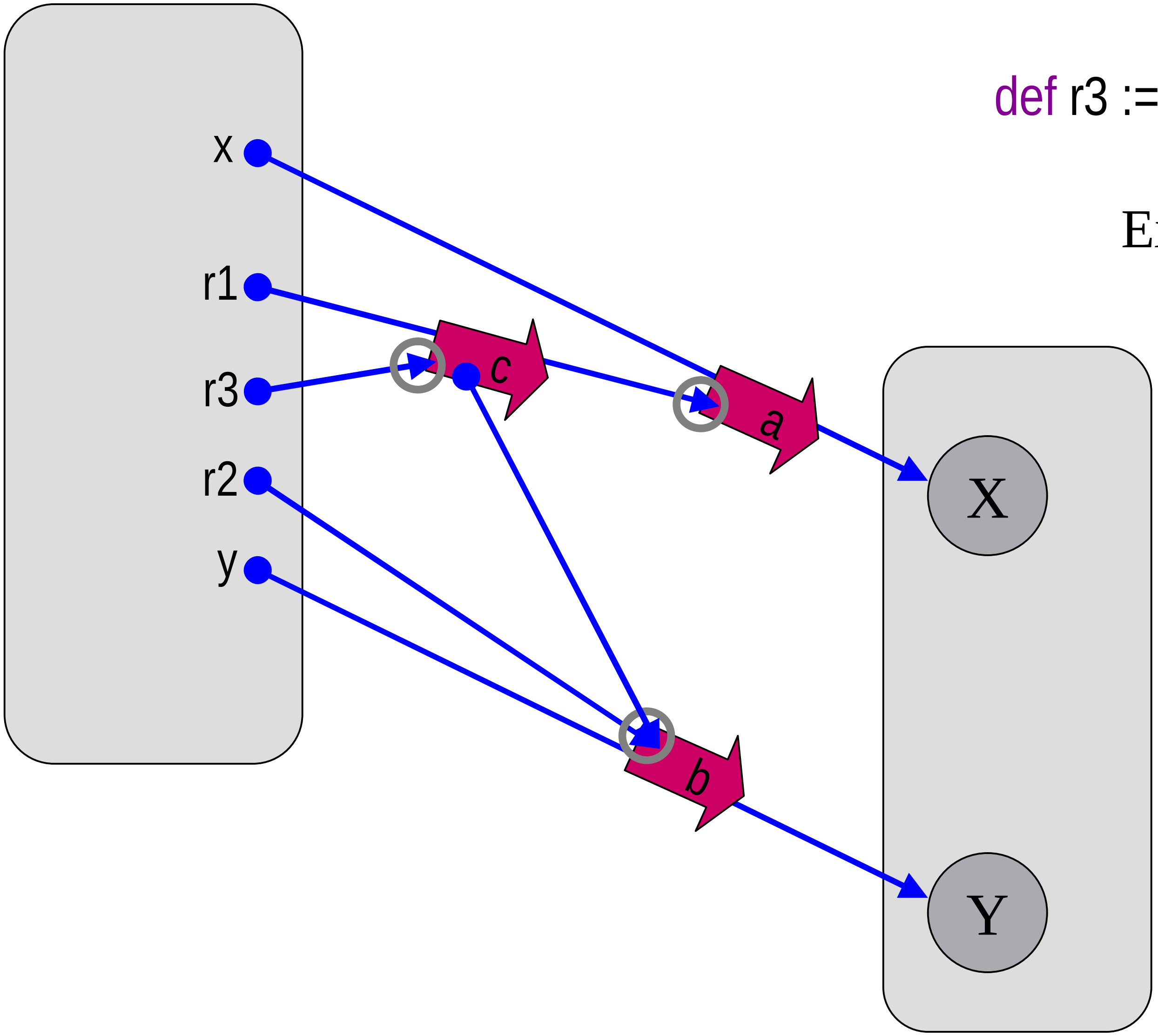


Bob says:

carol <- z()

FIFO <= E-ORDER <= CAUSAL
enforced by protocol

… with distant machines …



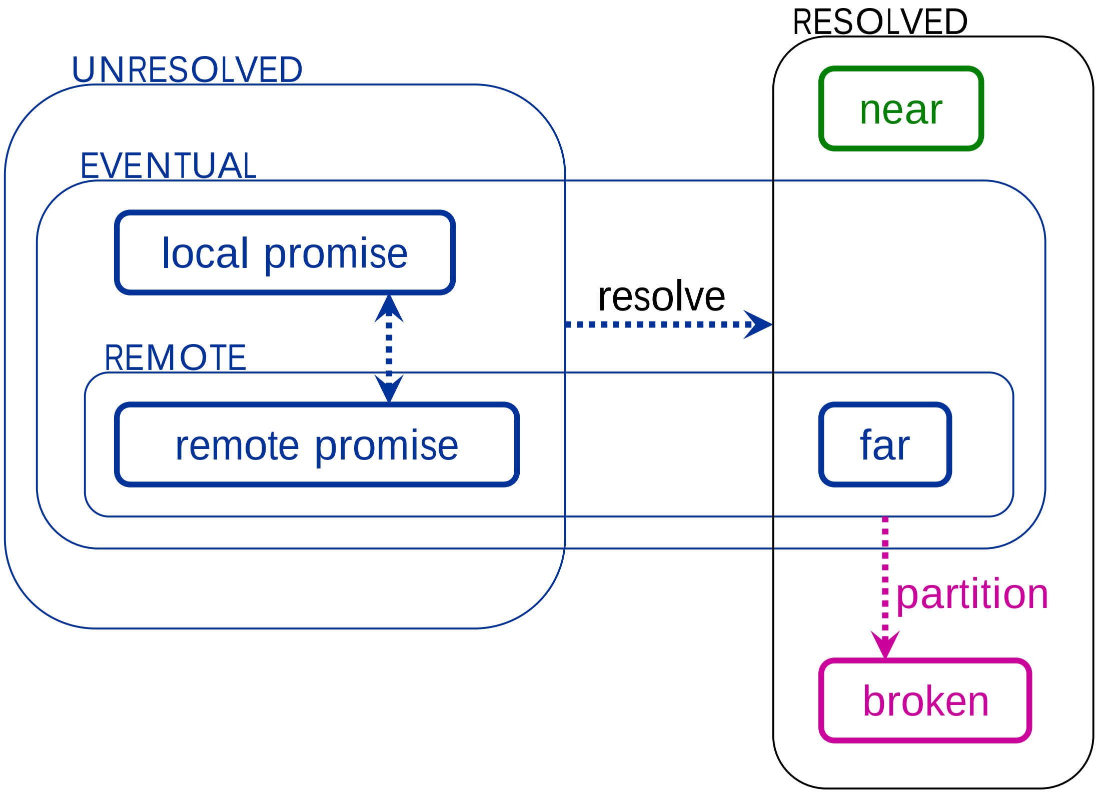
```
def r3 := (x <- a()) <- c(y <- b())
```

Expands to…

```
def r1 := x <- a()
def r2 := y <- b()
def r3 := r1 <- c(r2)
```

Messages always move towards arrowhead.

… that may not be reachable.



Near: “.” & “<-”

local objects

Eventual: only “<-”

promise or remote

lock-free atomicity

Broken: complains

Reify partition

NaN-like contagion

Unresolved Promises

Results of “<-”

pipeline messages

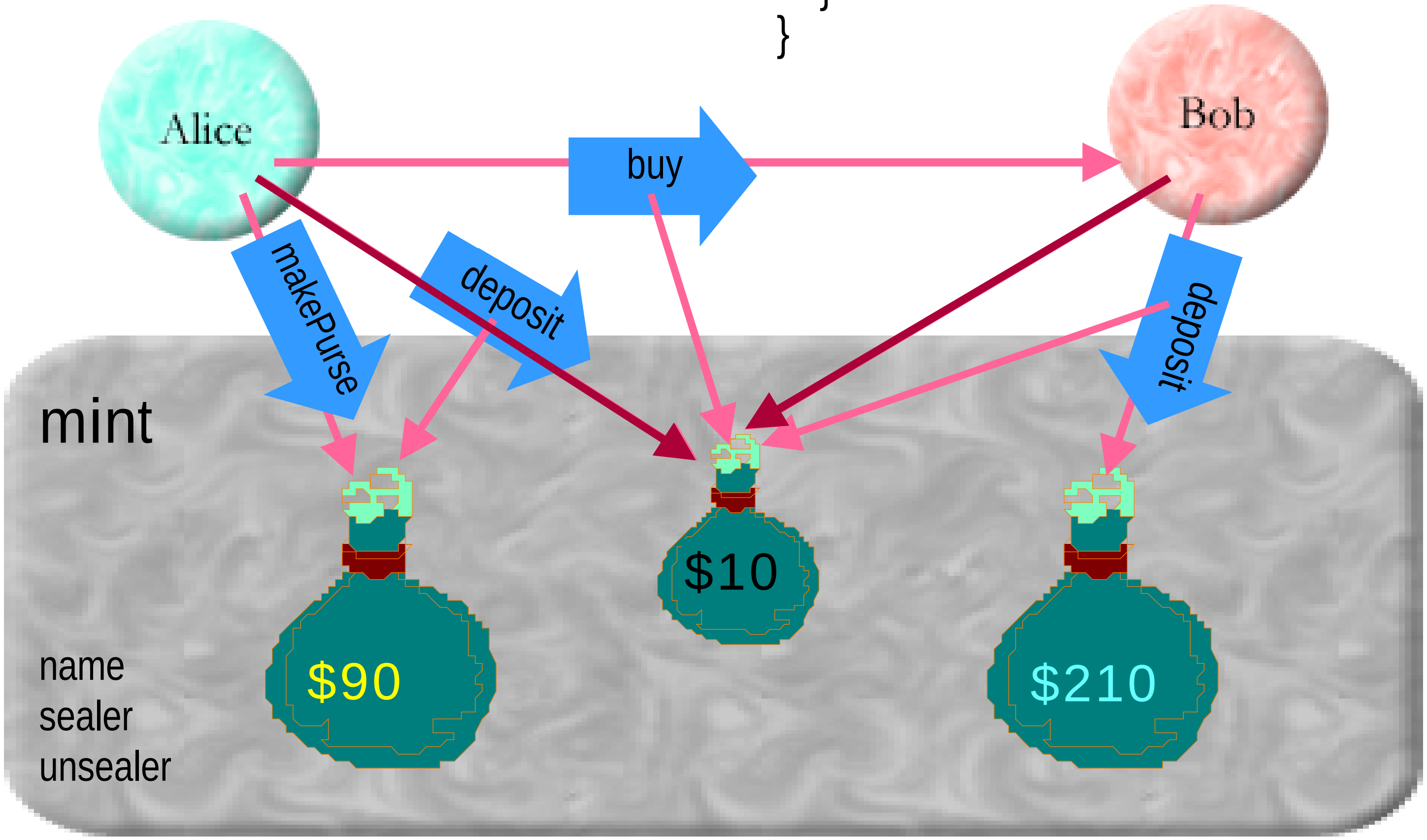
Off-line Caps (unshown)

Reify right to reconnect

Example: Alice pays Bob

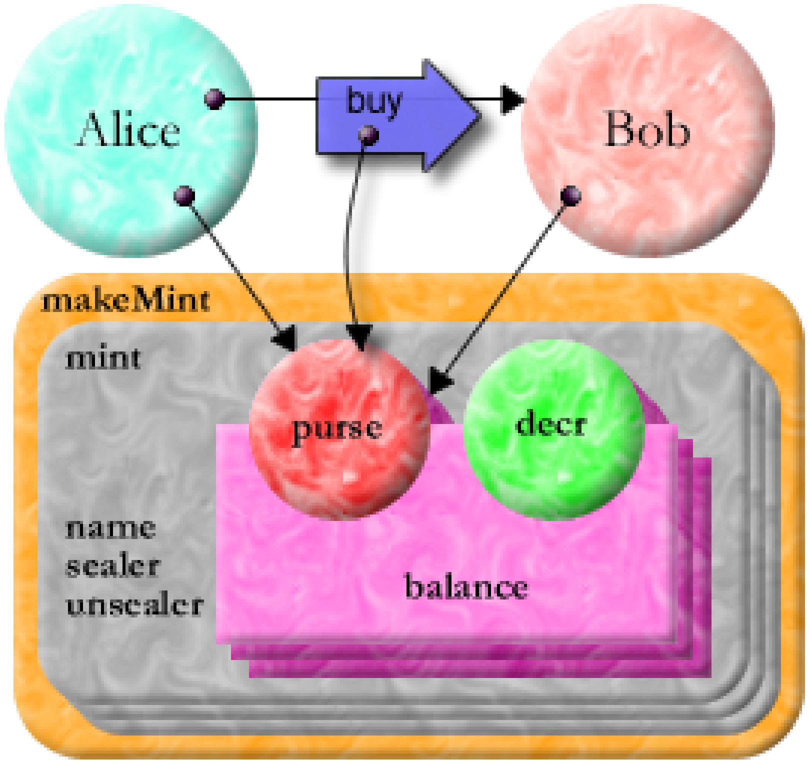
```
def payment := myPurse <- makePurse()
payment <- deposit(10, myPurse)
bob <- buy(..., payment)
```

```
when (payment) -> ... {
    when (myPurse <- deposit(10, payment)) ... {
        ... # dispense value
    }
}
```



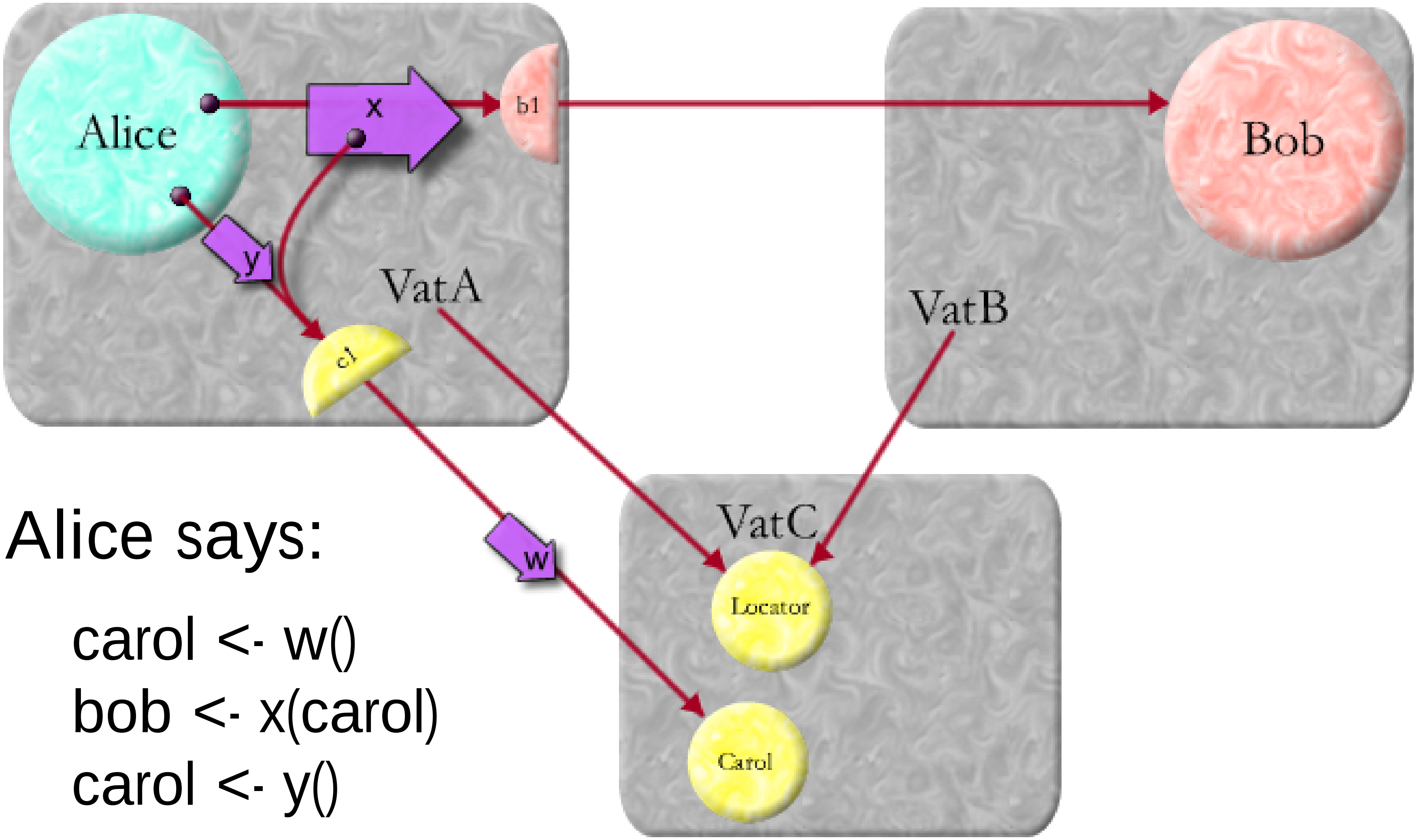
Distributed Secure Money in E

No explicit crypto



```
def makeMint(name :String) :any {
    def [sealer, unsealer] := makeBrandPair(name)
    def mint {
        to makePurse(var balance :(int >= 0)) :any {
            def decr(amount :(0..balance)) :void {
                balance -= amount
            }
            def purse {
                to getBalance() :int   { return balance }
                to makePurse() :any { return mint.makePurse(0) }
                to getDecr()       :any { return sealer.seal(decr) }
                to deposit(amount :int, src) :void {
                    unsealer.unseal(src.getDecr())(amount)
                    balance += amount
            }  }
            return purse
    }  }
    return mint
}
```

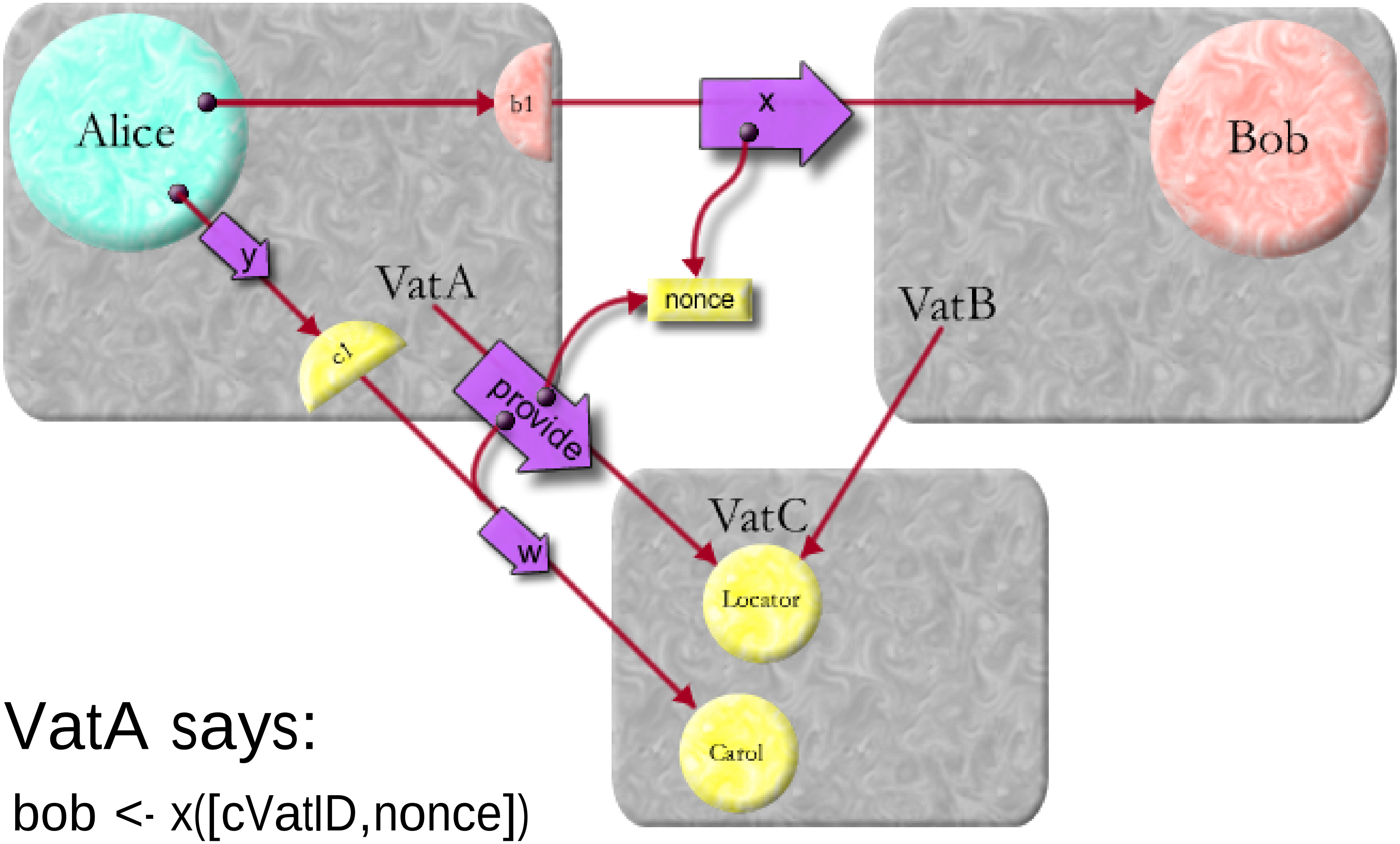
Enforcing E-Order



Alice says:

```
carol <- w()
bob <- x(carol)
carol <- y()
```

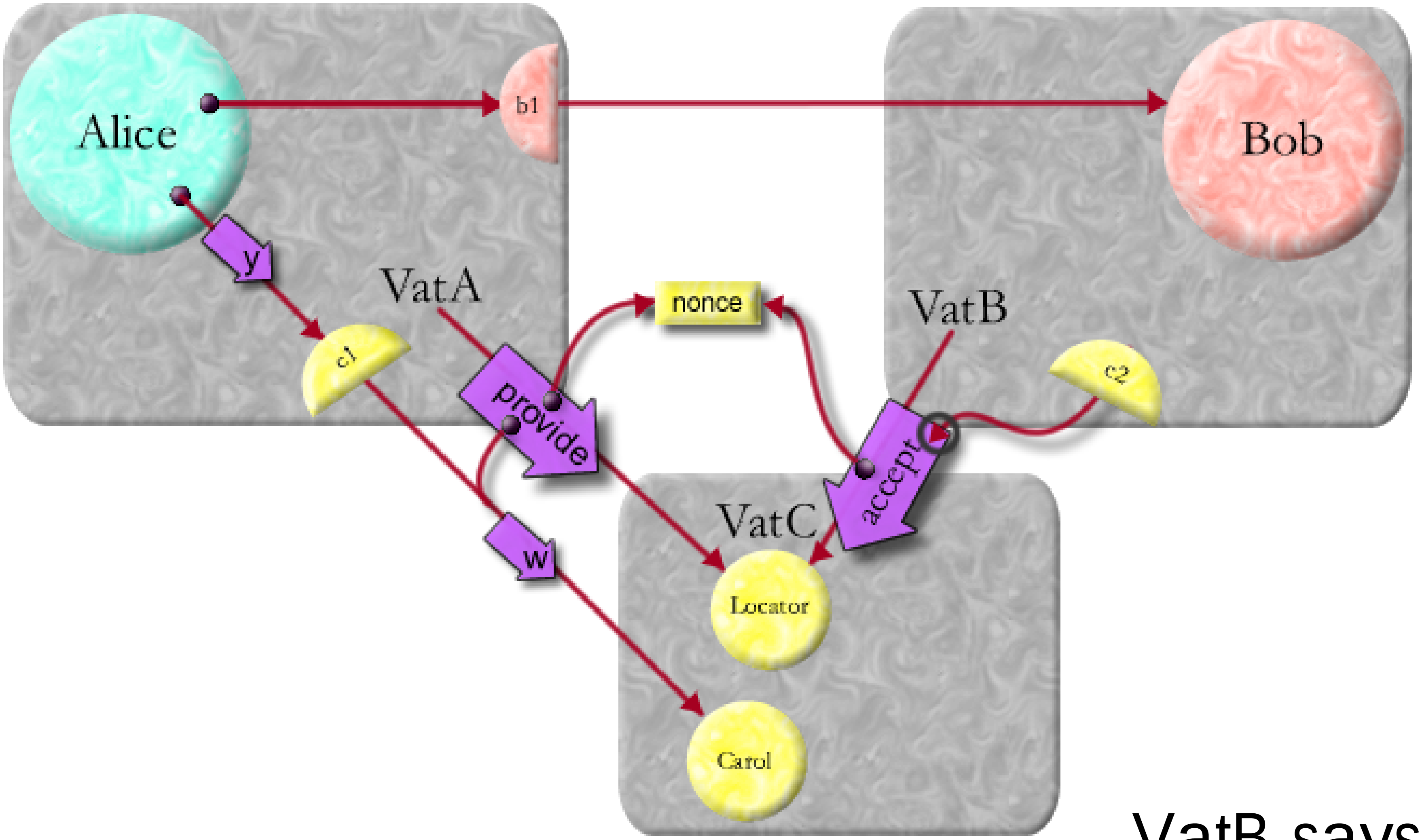
Enforcing E-Order



VatA says:

bob <- x([cVatID,nonce])
cLocator <- provide(carol,nonce)

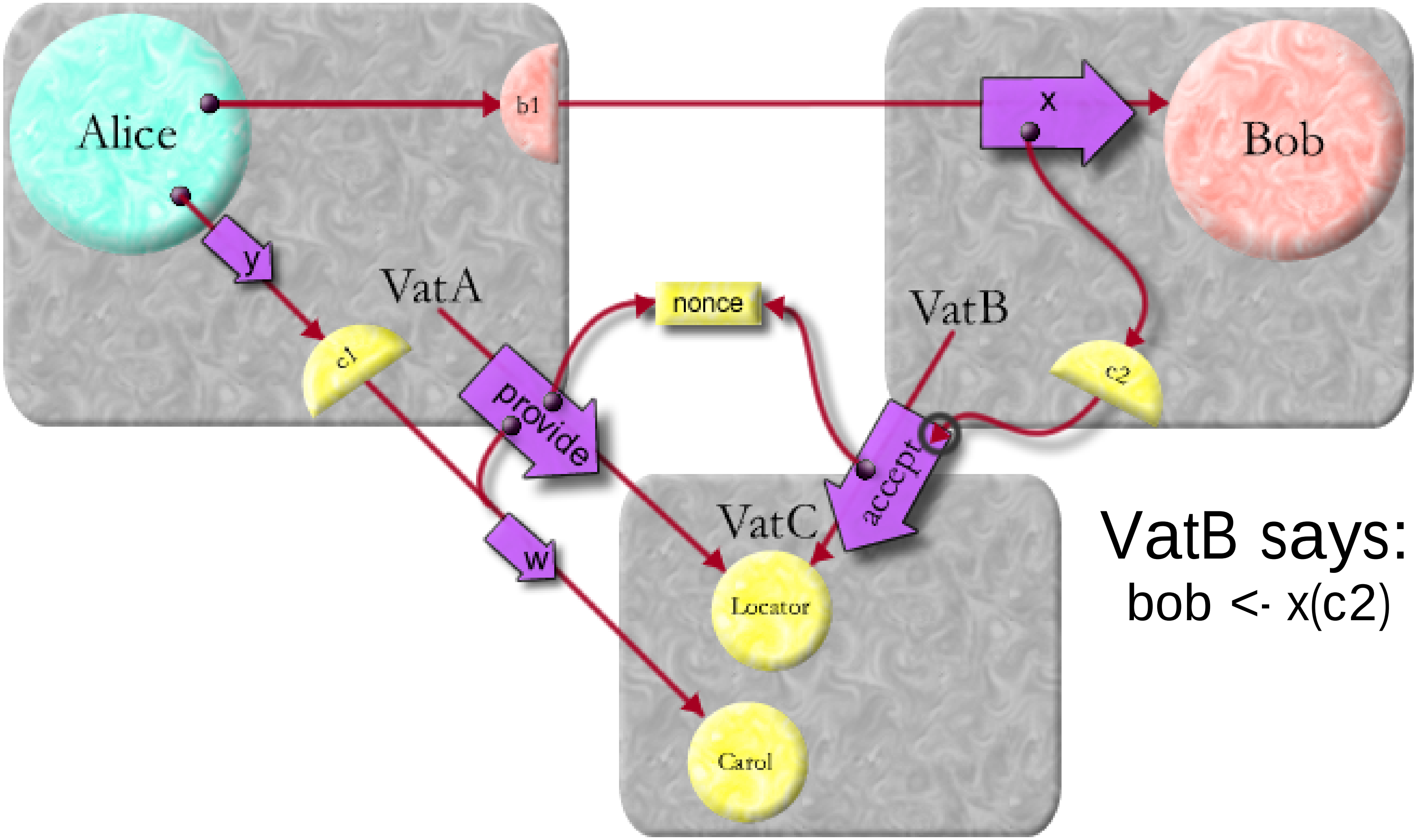
Enforcing E-Order



VatB says:

def c2 := cLocator <- accept(nonce)

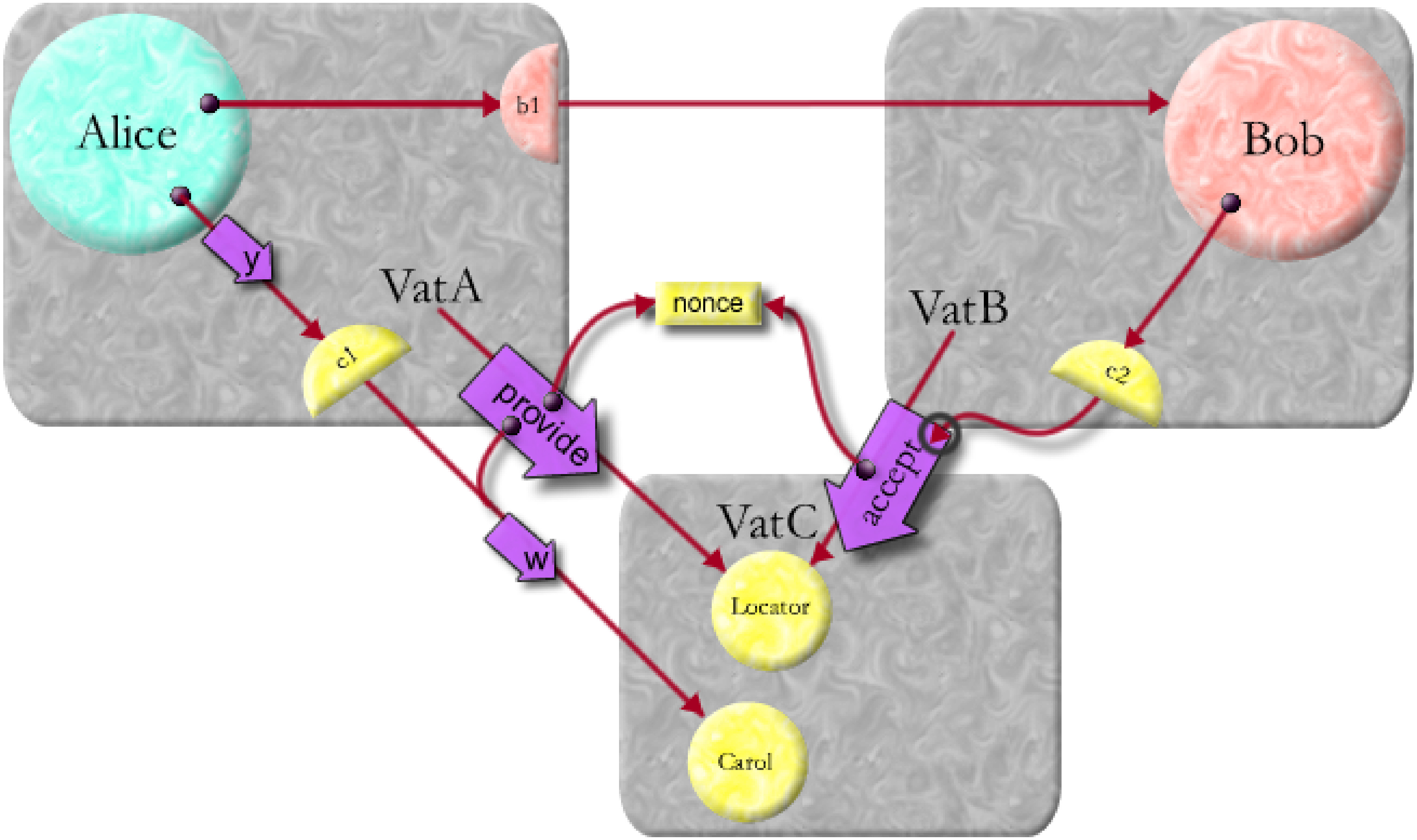
Enforcing E-Order



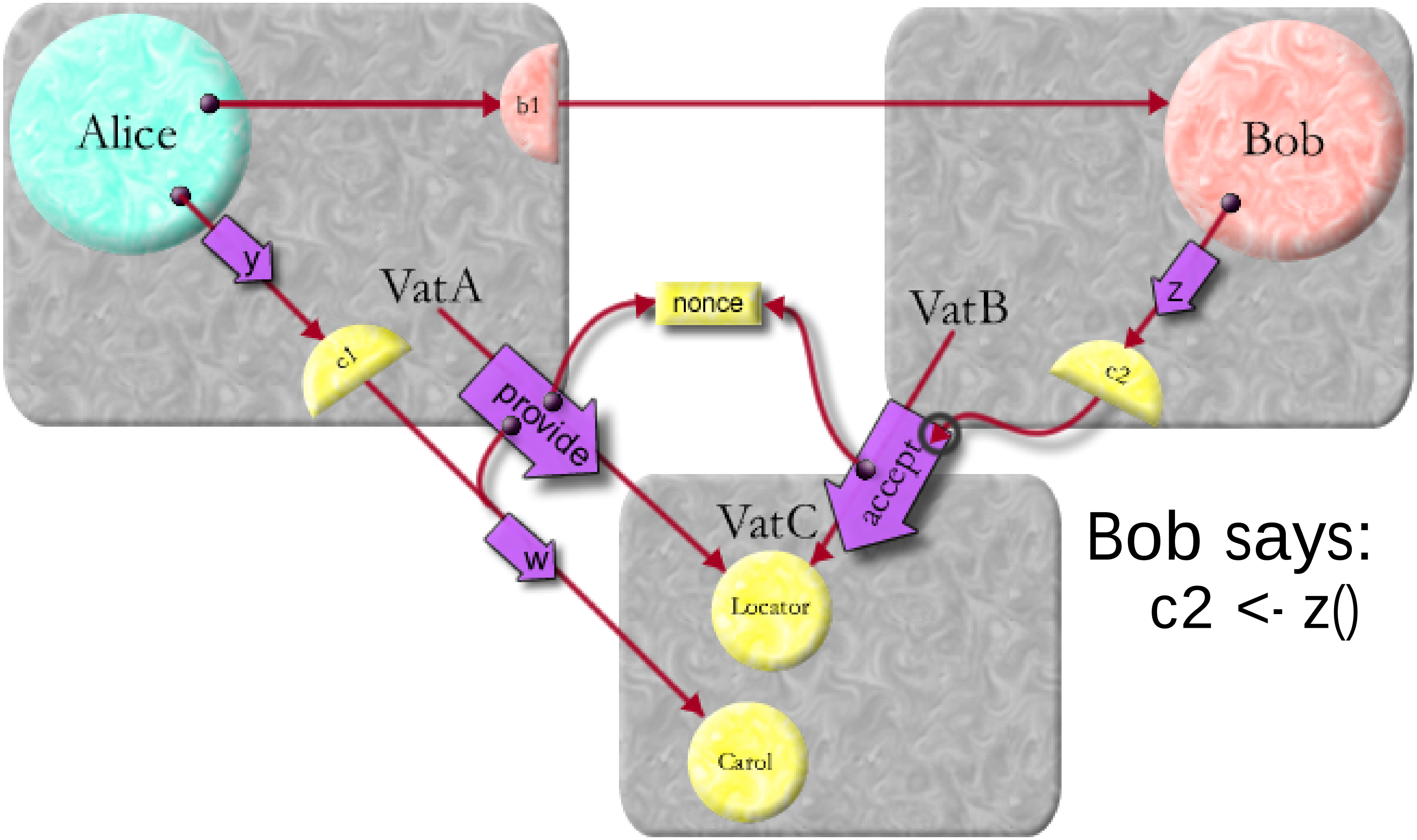
VatB says:

bob <- x(c2)

Enforcing E-Order

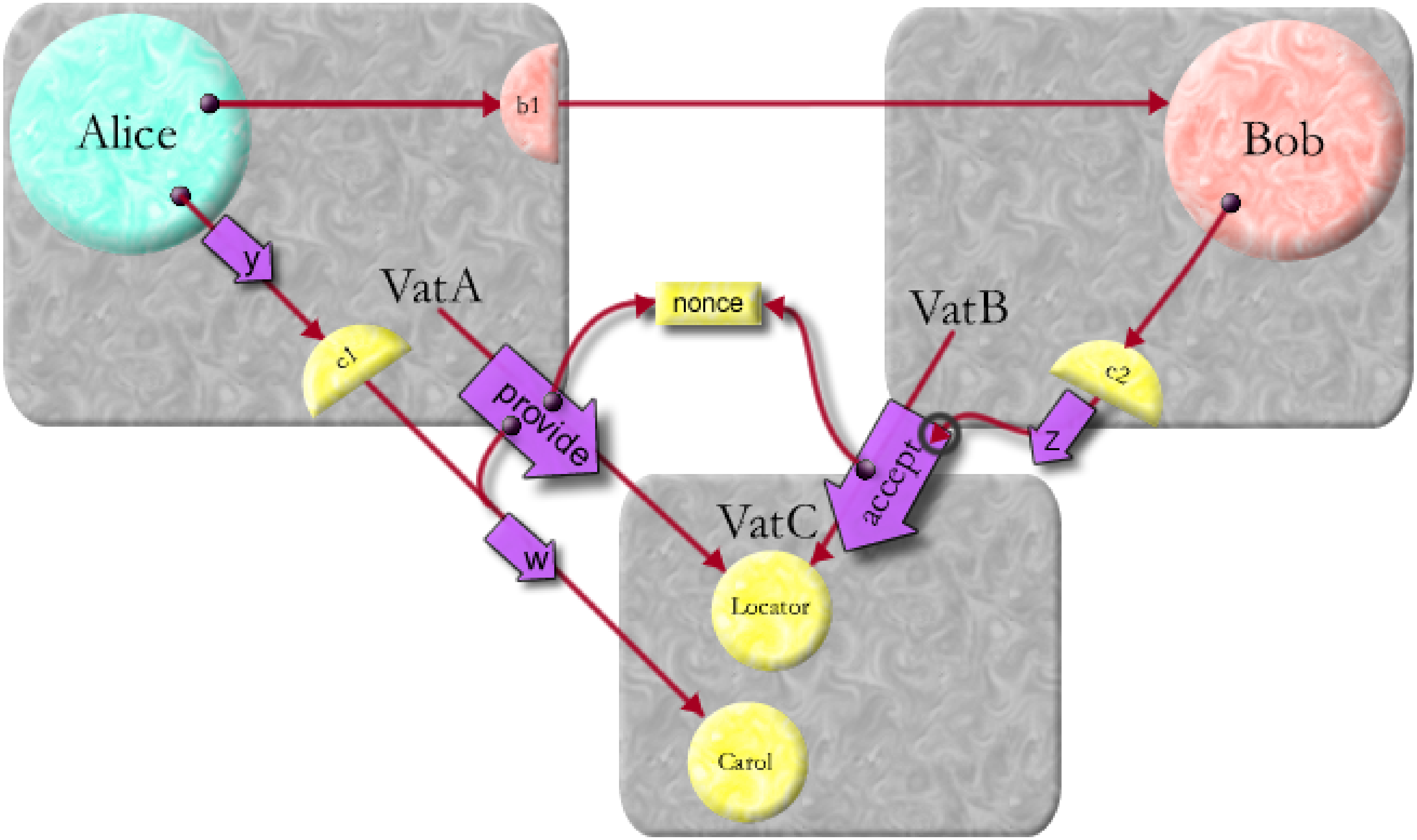


Enforcing E-Order

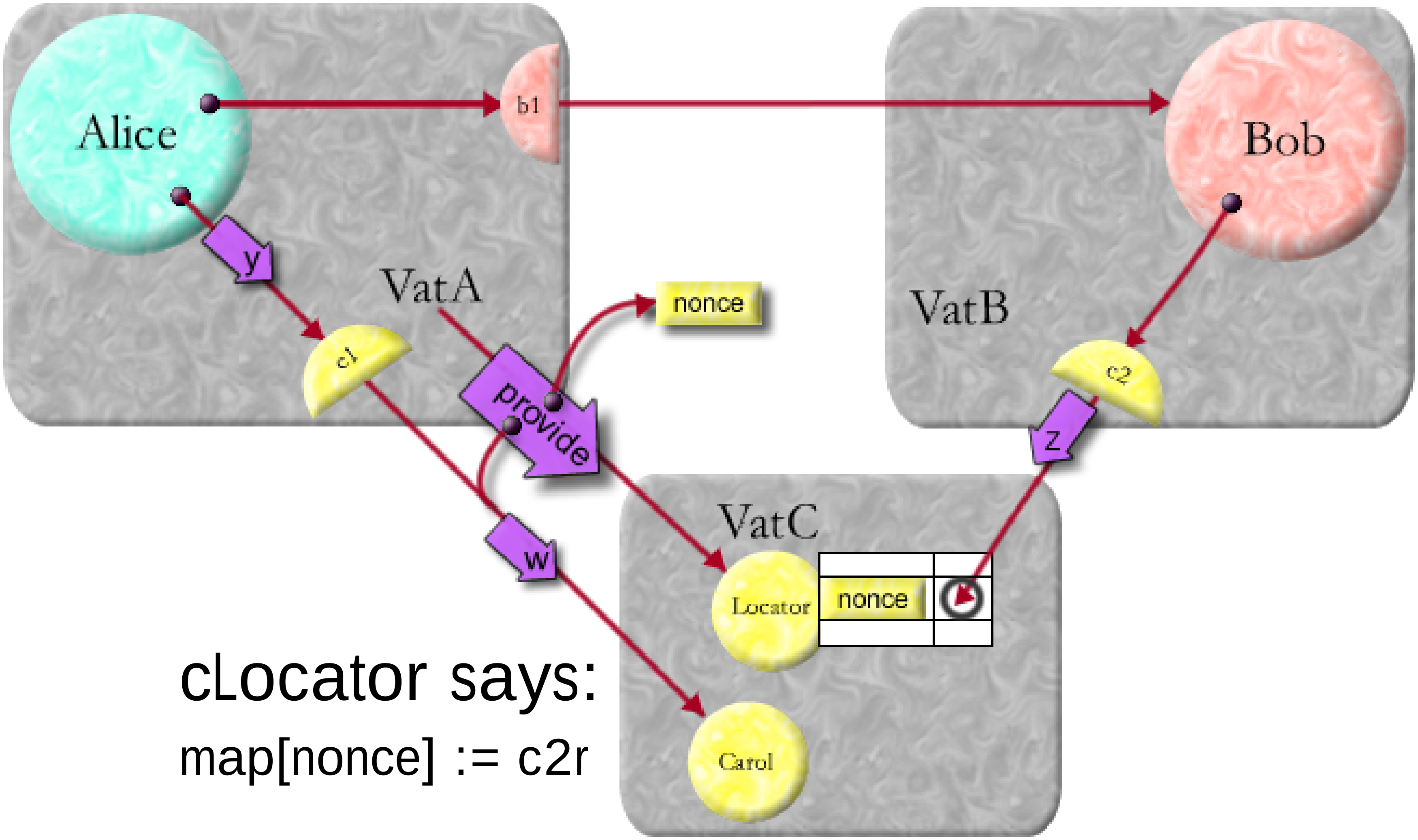


Bob says:
c2 <- z()

Enforcing E-Order



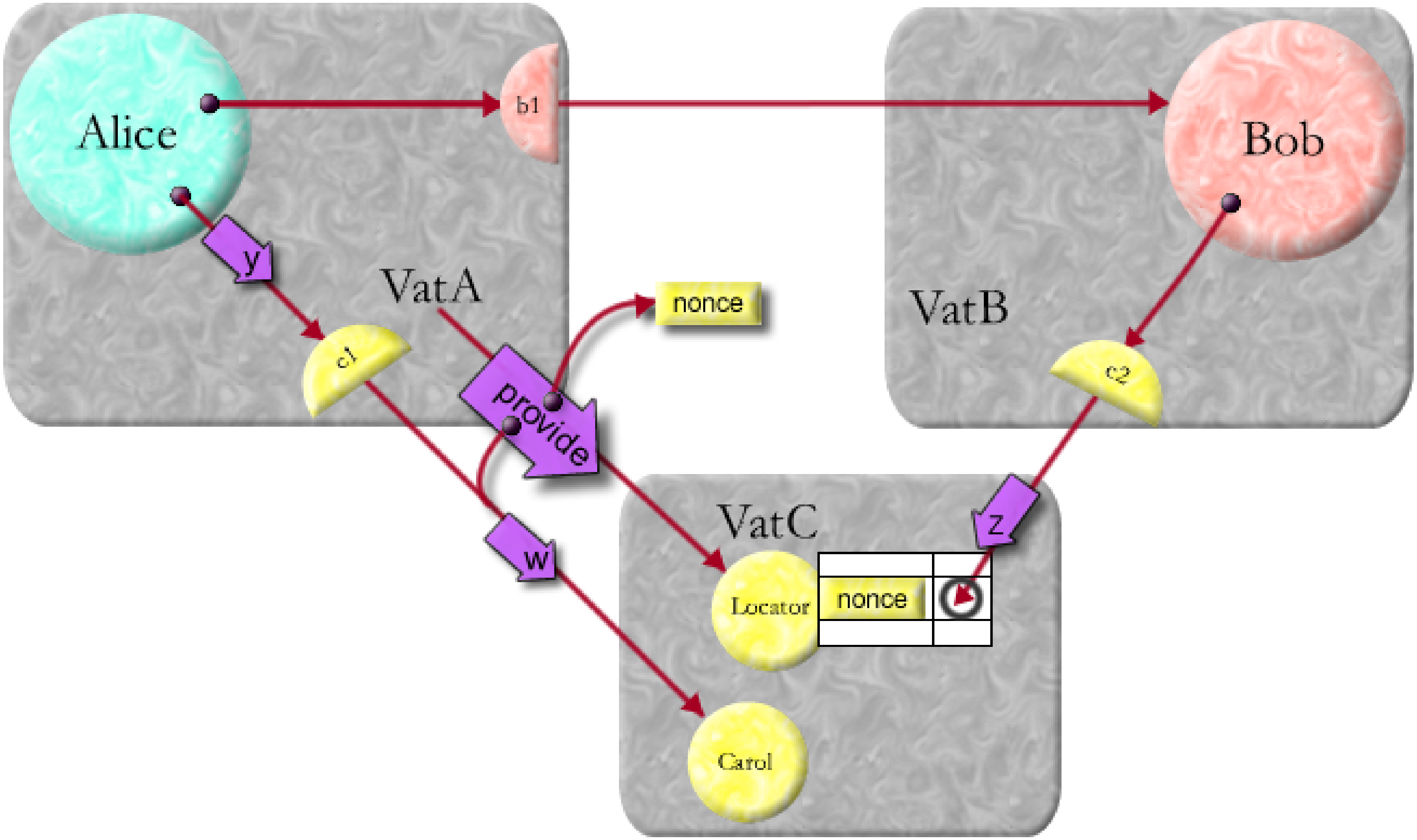
Enforcing E-Order



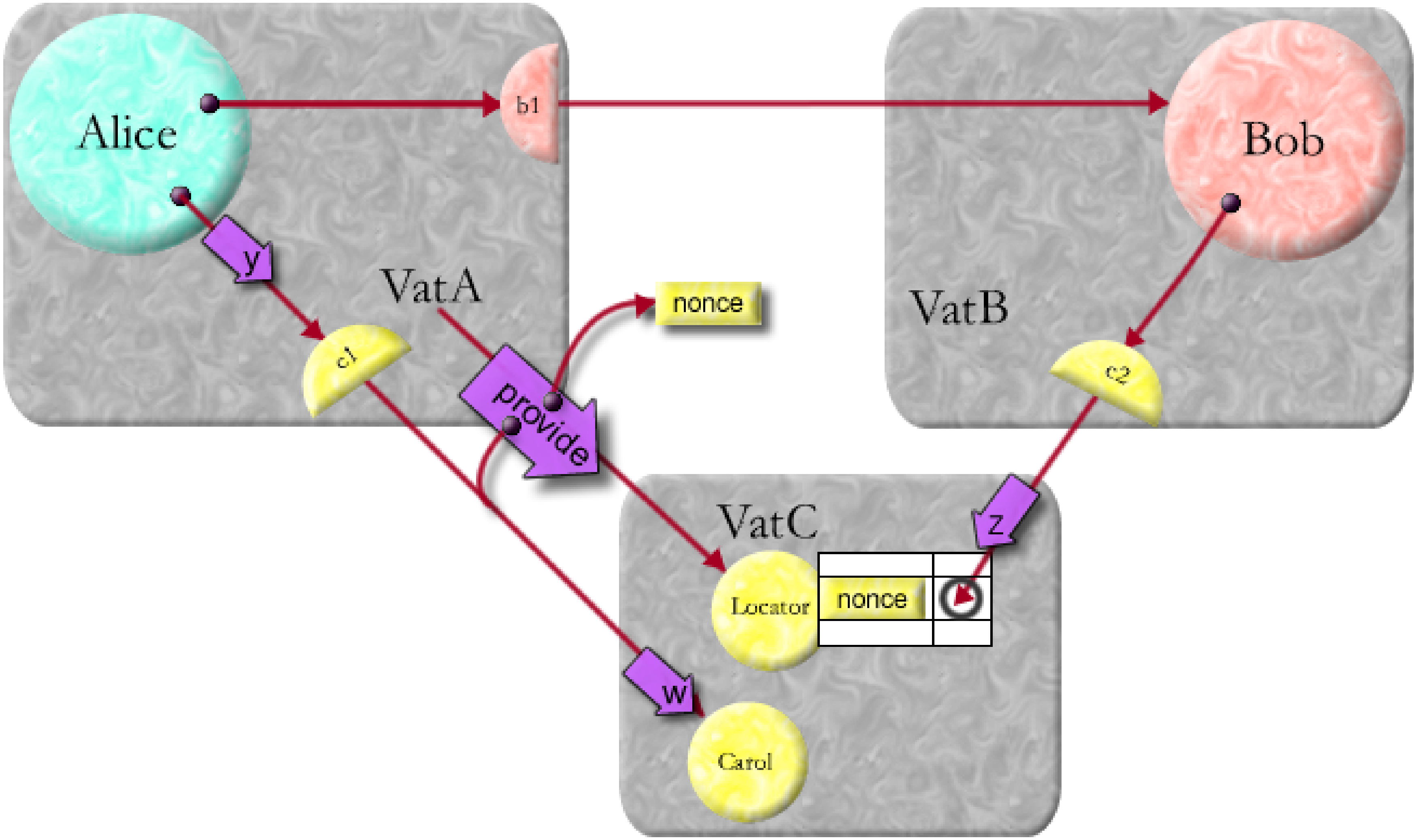
cLocator says:

map[nonce] := c2r

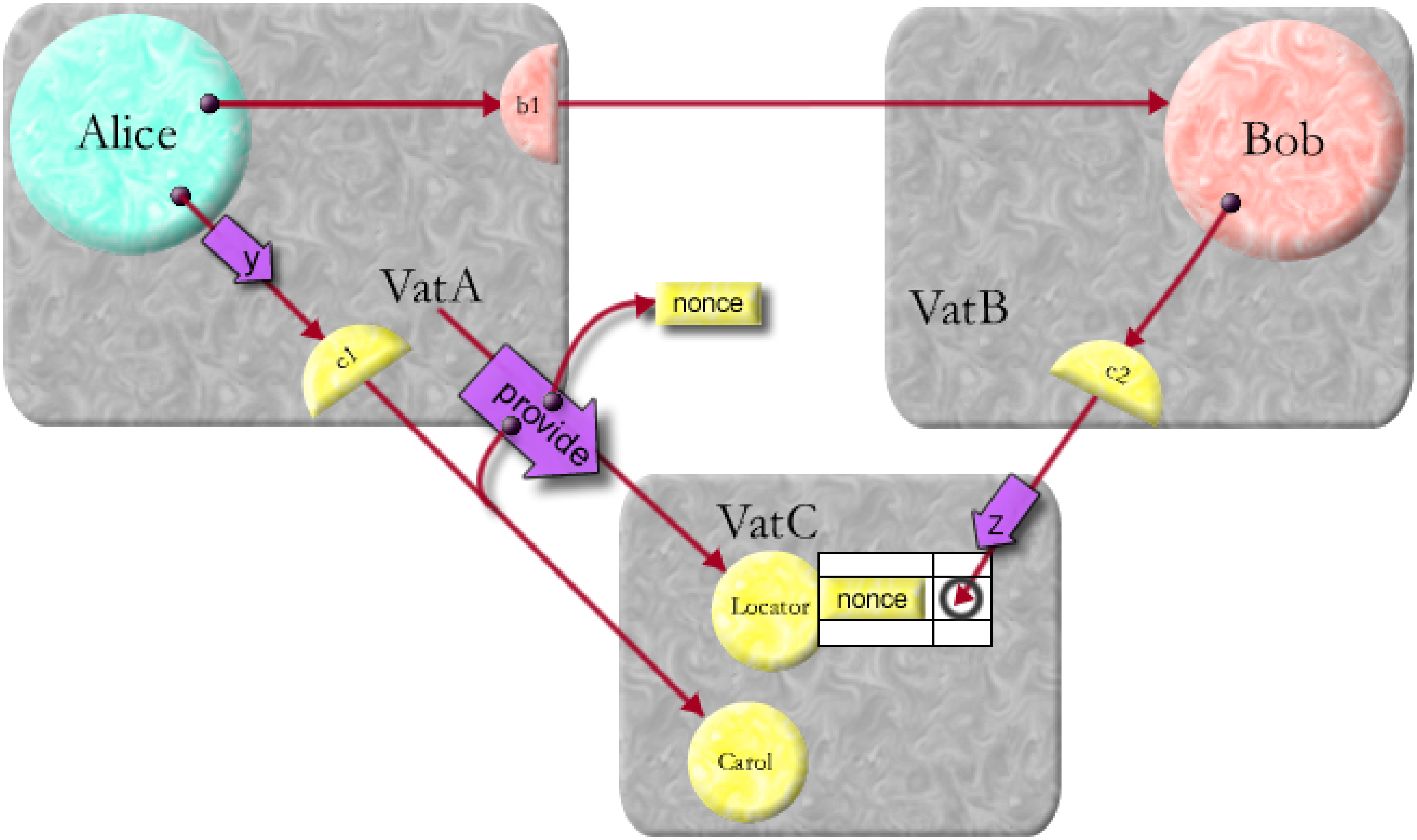
Enforcing E-Order



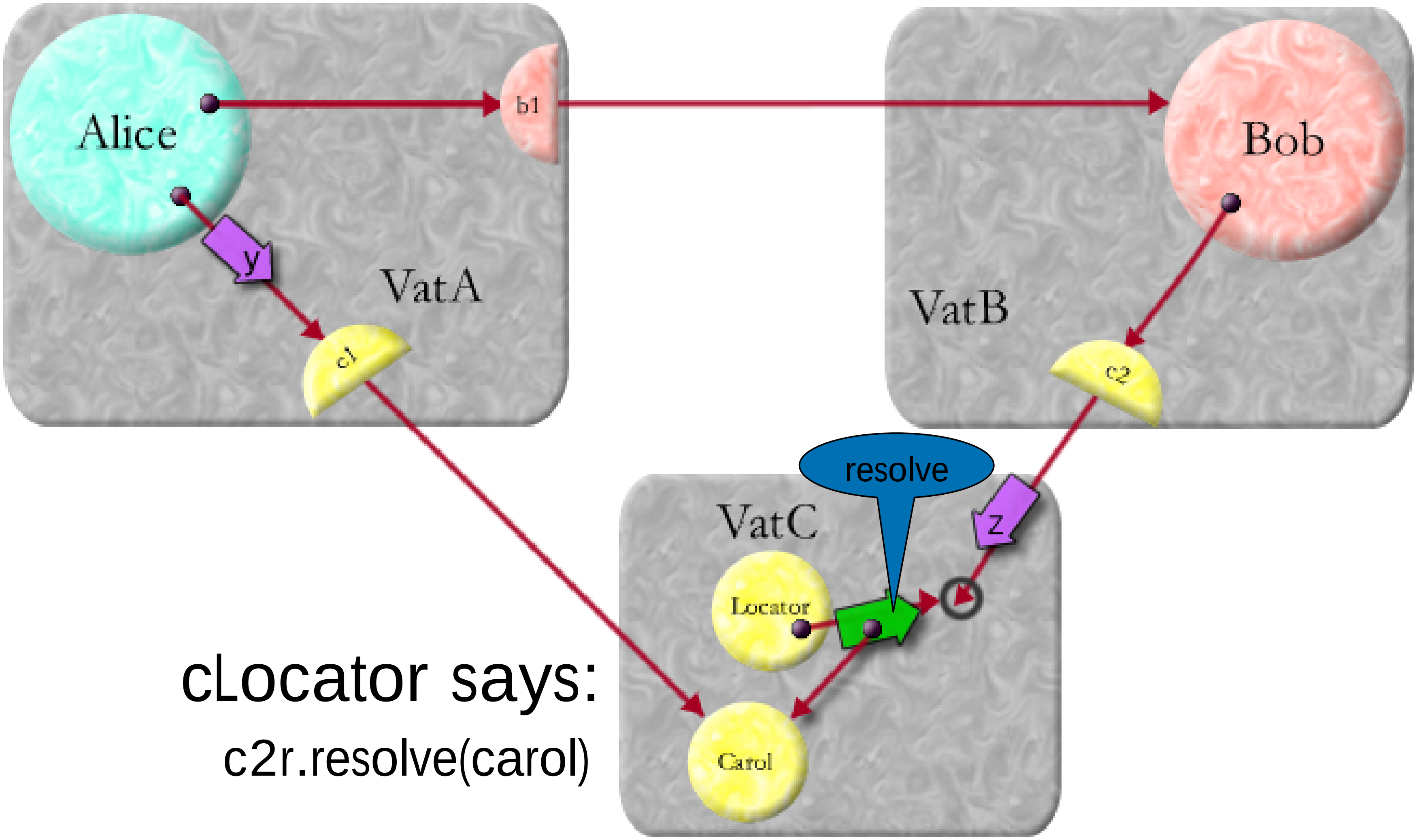
Enforcing E-Order



Enforcing E-Order



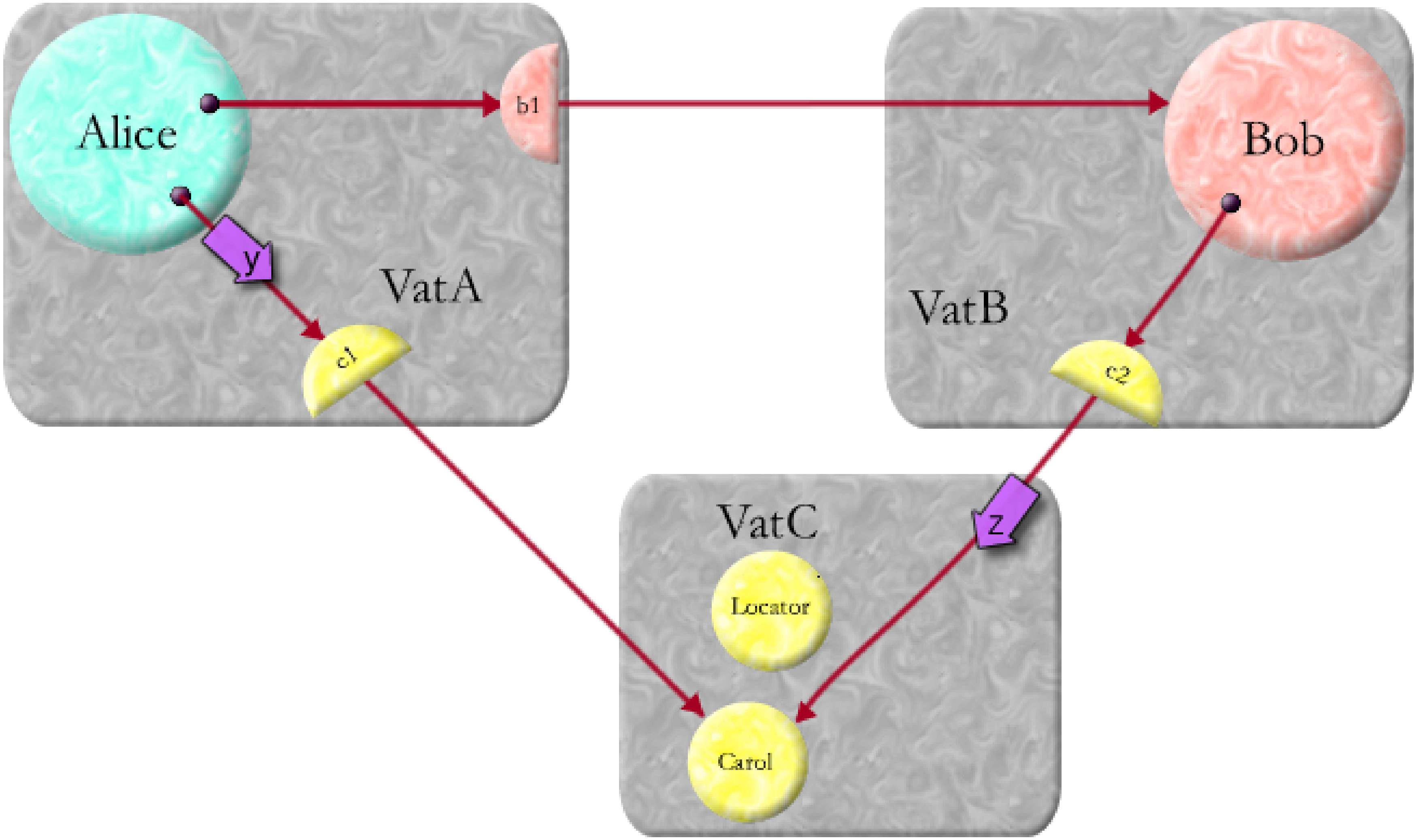
Enforcing E-Order



cLocator says:

c2r.resolve(carol)

Enforcing E-Order


Alice
b1
Bob
y
VatA
VatB
c1
c2
z
VatC
Locator
Carol