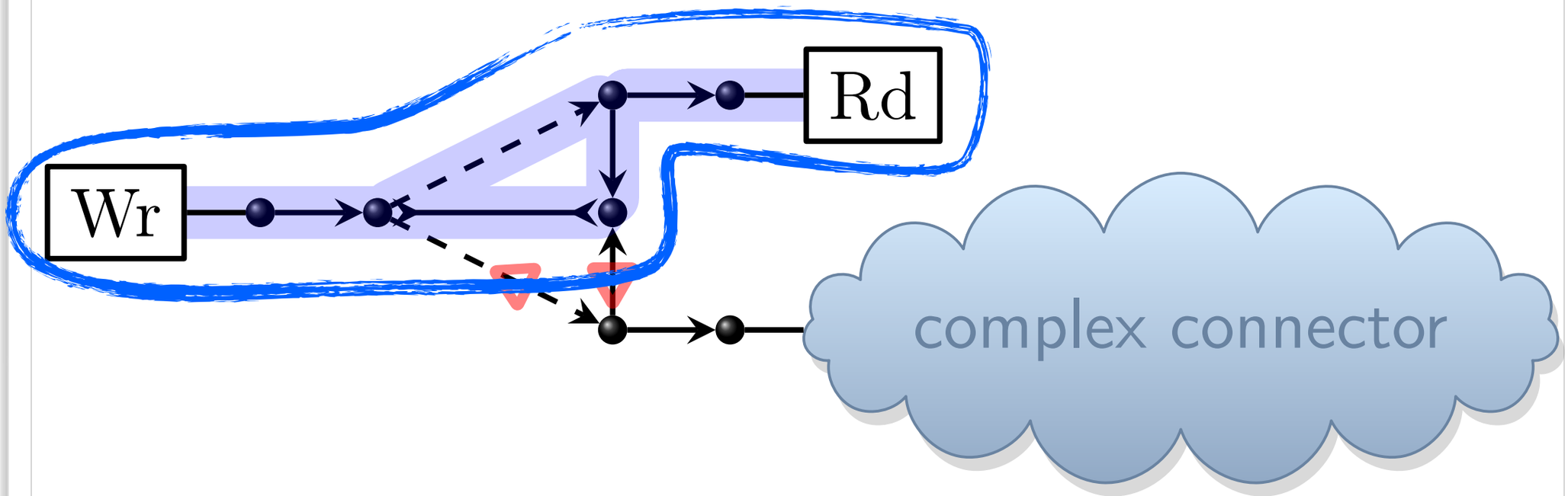
Architectural design: the coordination perspective

José Proença
HASLab - INESC TEC & UM
Arquitectura e Cálculo 2015-16



Locality (concurrency)

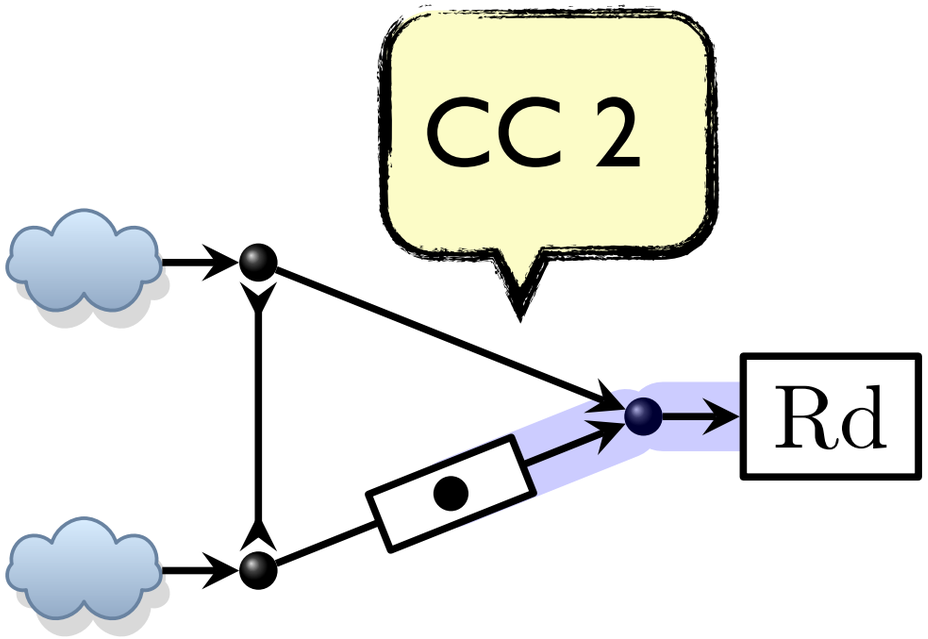
Motivation

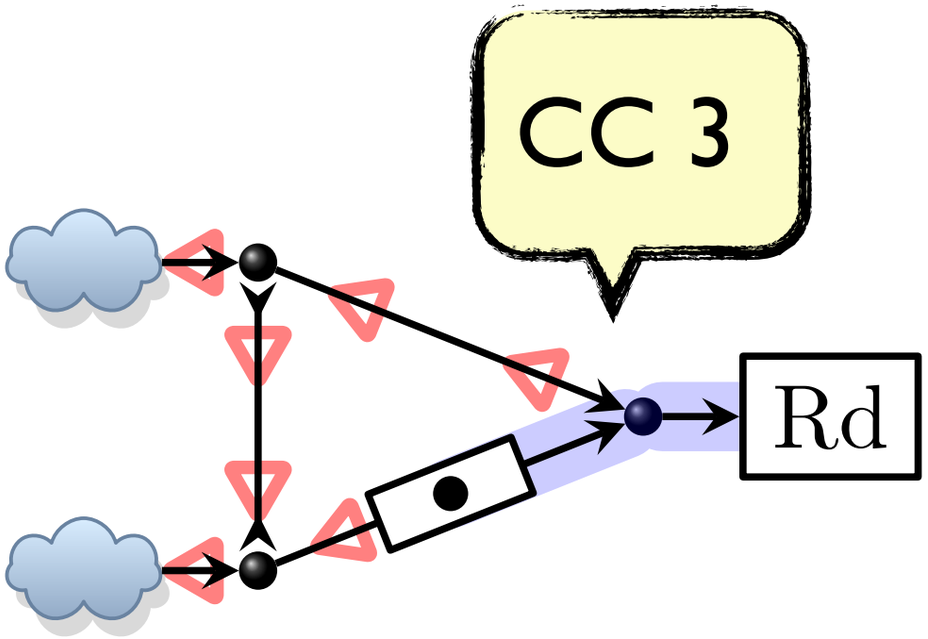
- Connector colouring is not optimal for distributed systems.
- All-or-nothing – all channels are needed to decide where data goes.
- Need to identify local flows that are not composed with the full connector.
- Model context dependency

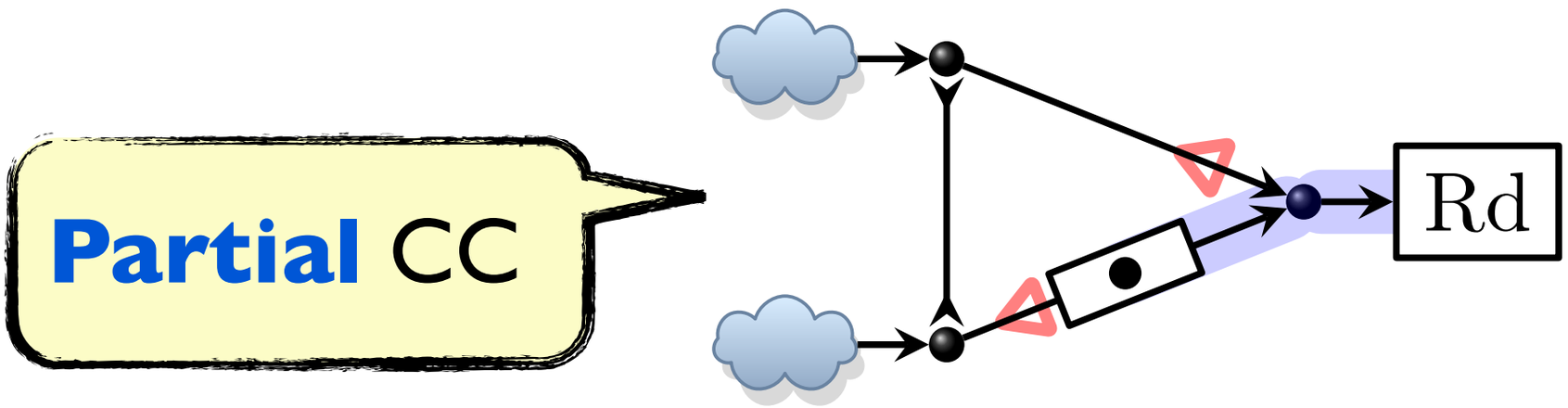
Problems

- 2 colours (or constraint automata):
  - assume primitives can make a *no-flow* step
- 3 colours:
  - cannot assume primitives have a no-flow colour – which direction would it be?
  - Idea?: add another no-flow colour, without direction, and assume all primitives have it...

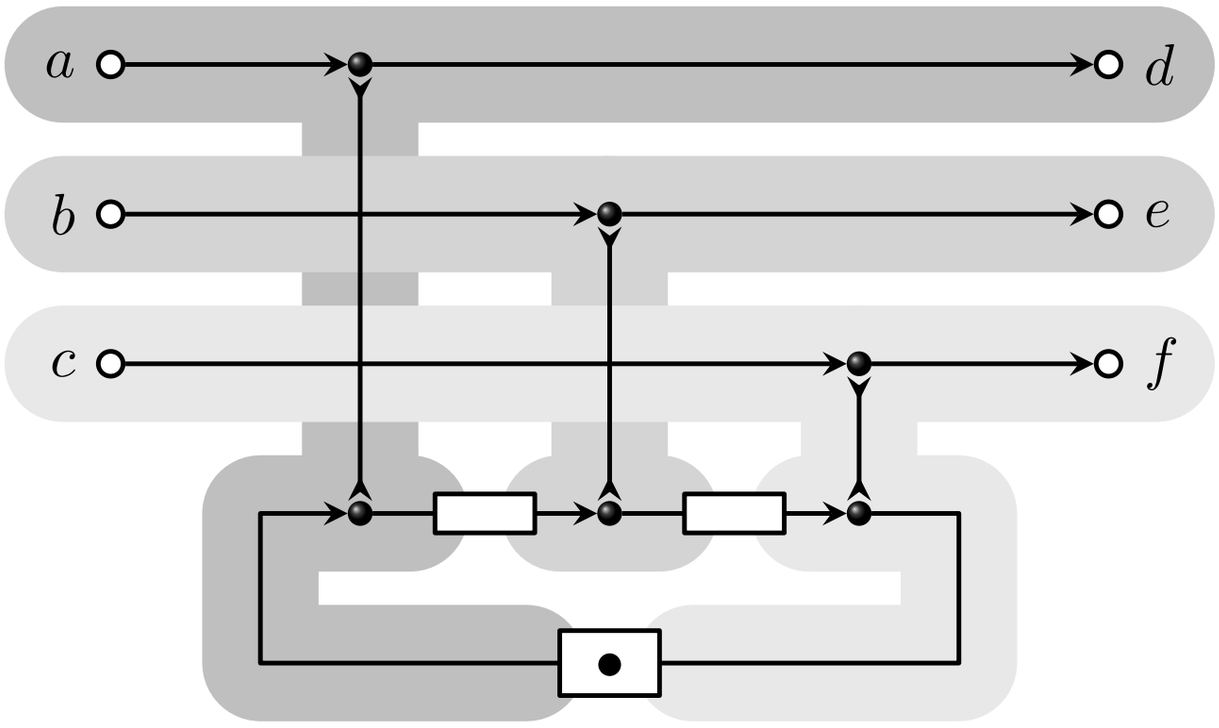
Example





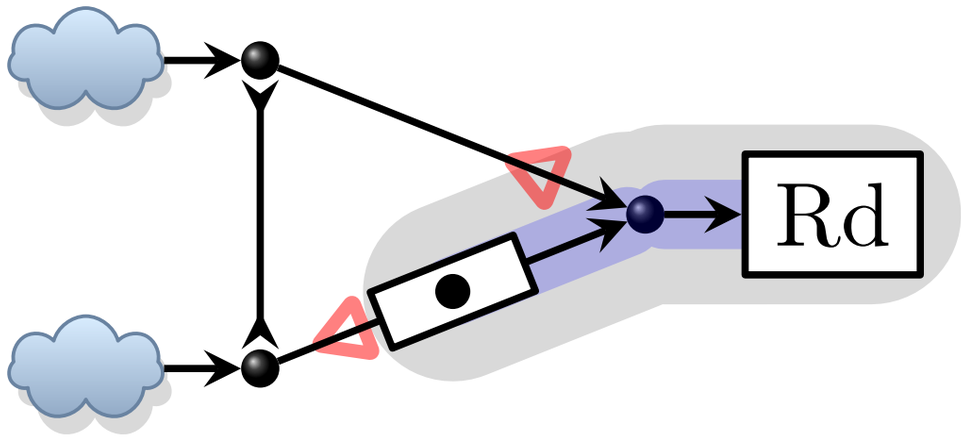


Synchronous regions



Static regions:
boundaries
=
FIFO's

Dynamic regions:
boundaries
=
GiveReason

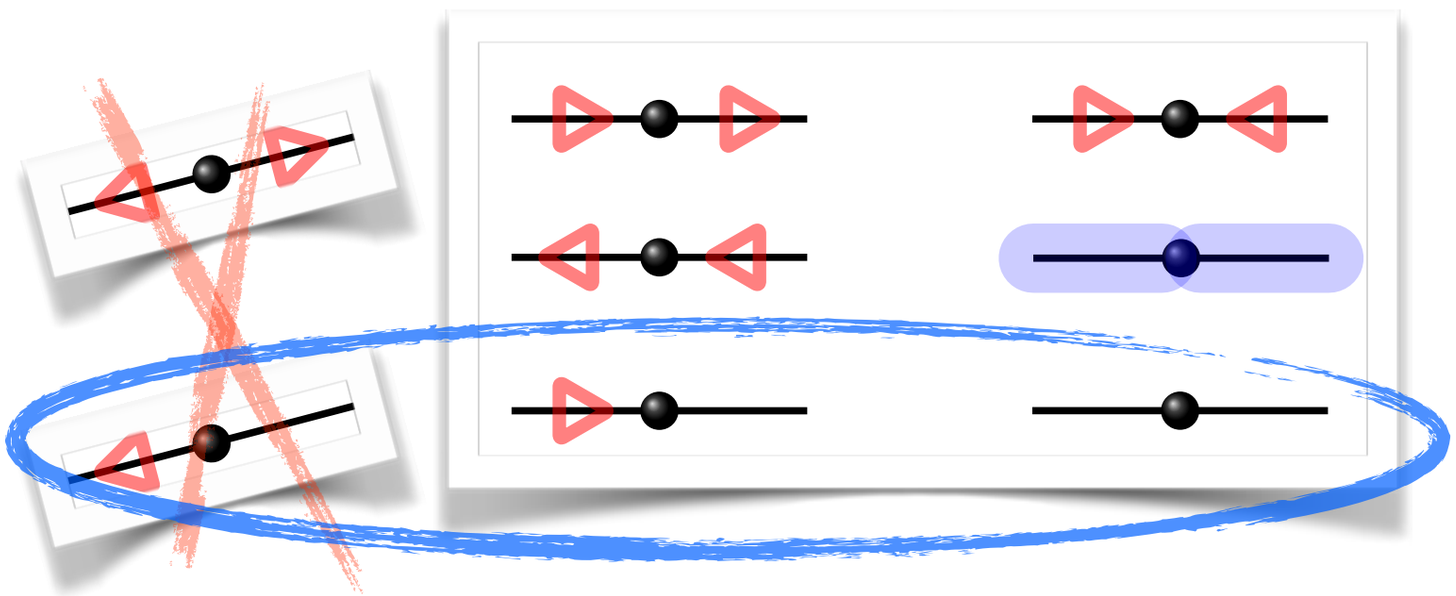


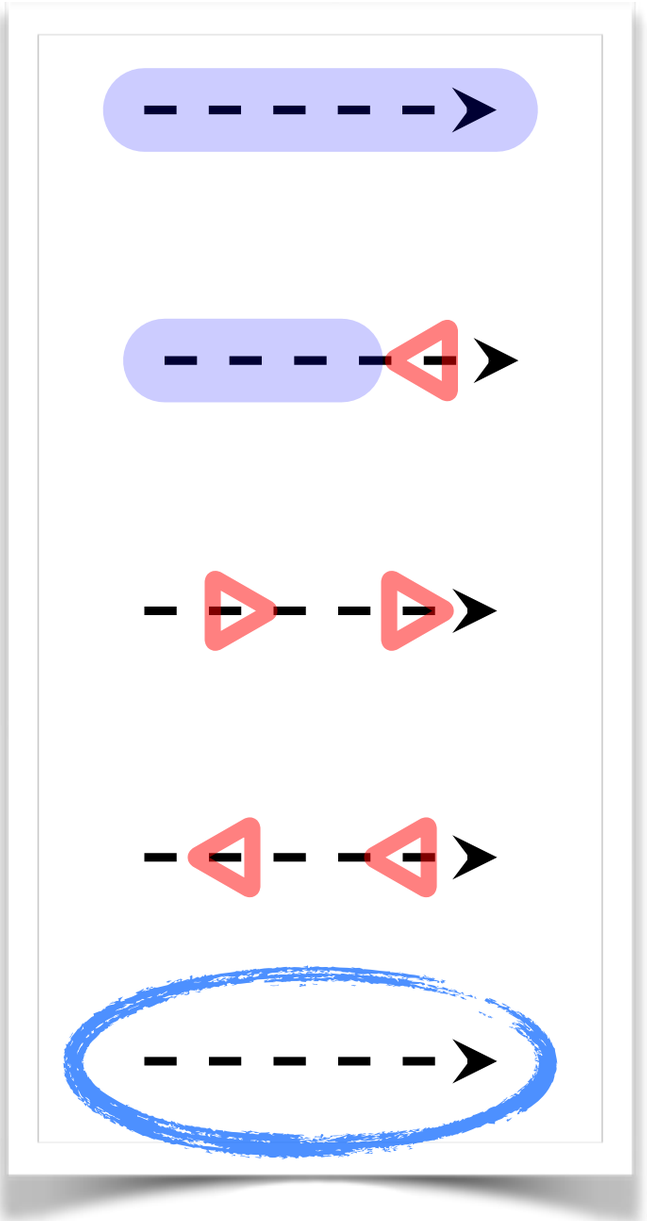
Partial connector colouring

- *Colouring*:

End → {Flow, GiveReason, GetReason}

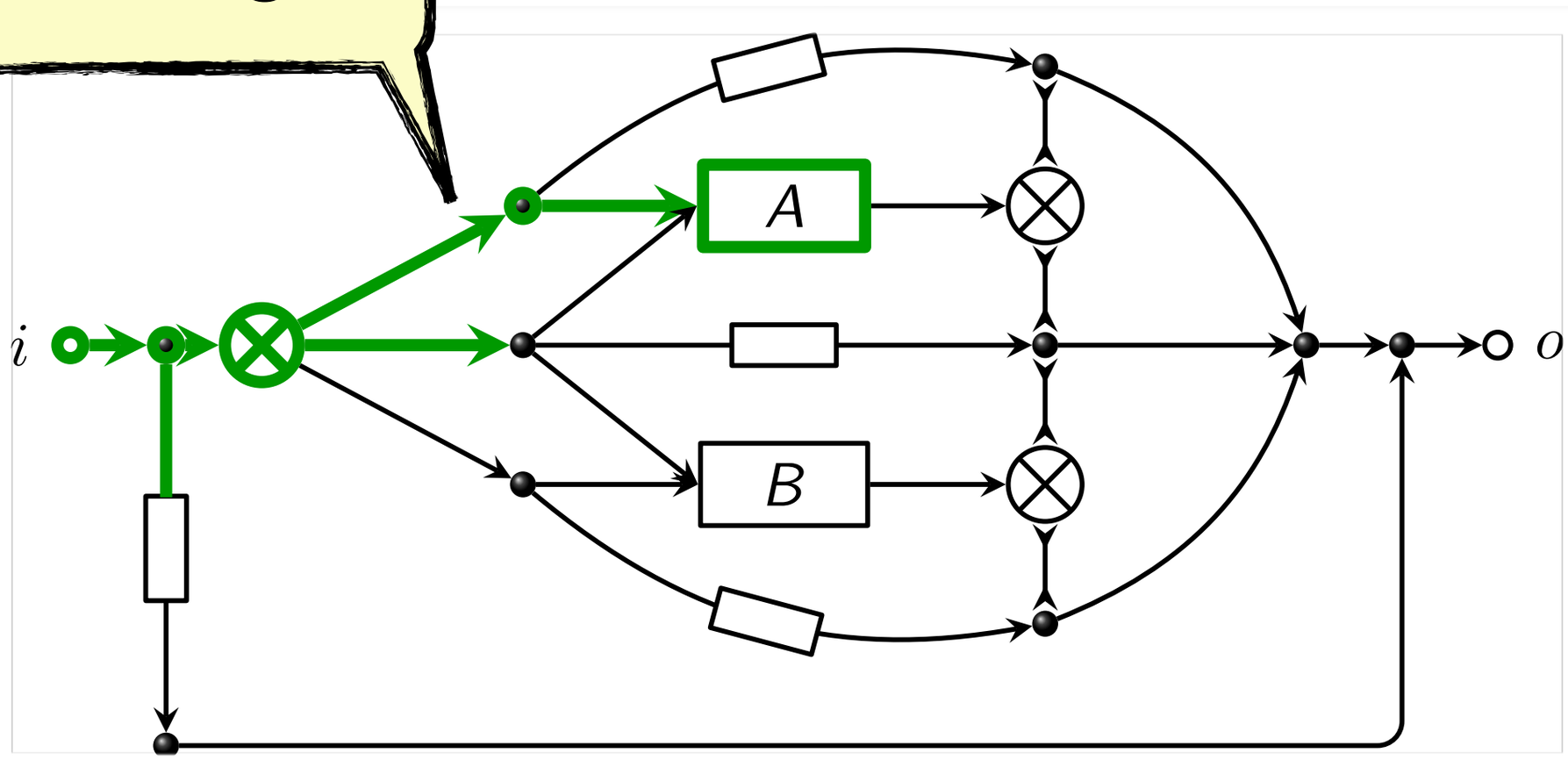
- *Composition* = matching colours:





In practice

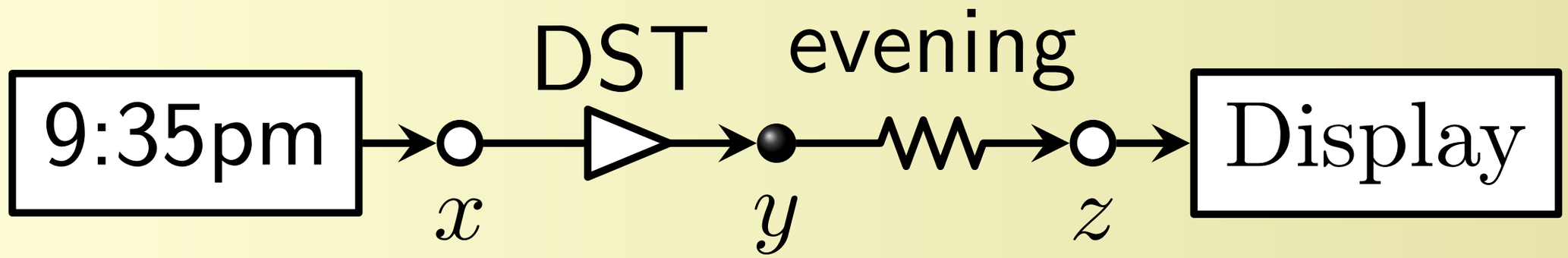
Shall I search now for a colouring?



Outline

1. Visual semantics for Reo
   - Connector colouring (CC)[1]
2. Locality (concurrency)
   - partial connector colouring (PCC)[2]
3. Constraints
   - SAT solving with data for Reo[3]

[1] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: Synchronisation and context dependency
[2] Dave Clarke and José Proença. Partial connector colouring
[3] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab, Channel-based coordination via constraint satisfaction
José Proença, Dave Clarke, Interactive interaction constraints



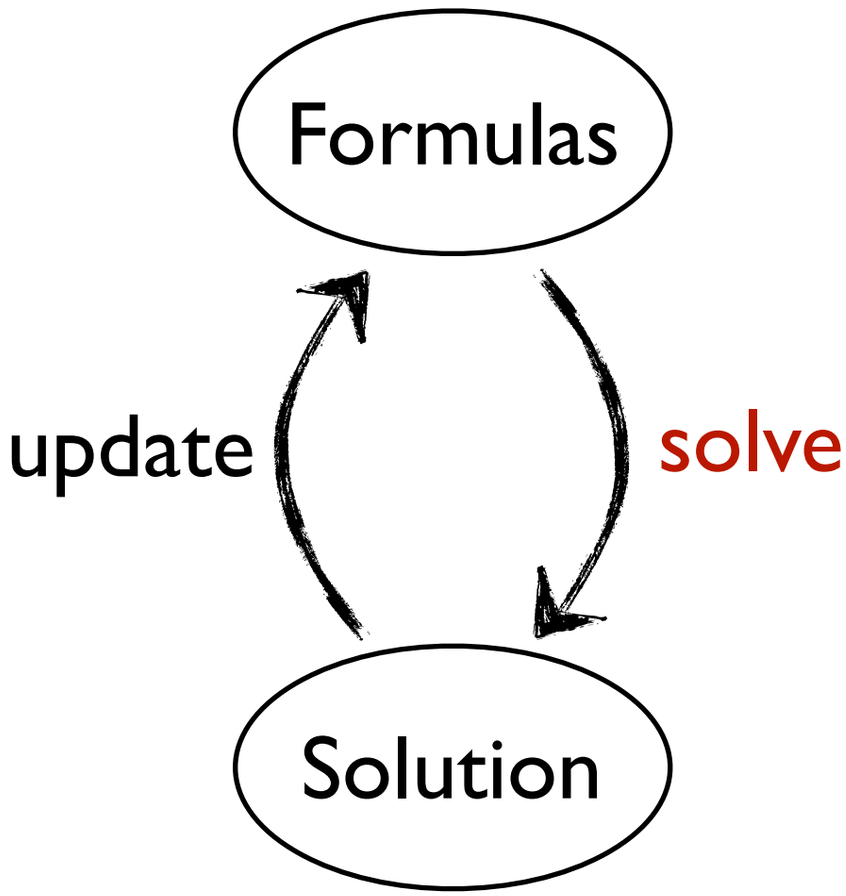
$x \to \widehat{x} := \mathsf{9{:}35pm}$ $\quad x \leftrightarrow y$ $\quad y \to \widehat{y} := \mathsf{DST}(\widehat{x})$

$(y \wedge \mathsf{evening}(\widehat{y})) \leftrightarrow z$ $\quad z \to \widehat{z} := \widehat{y}$

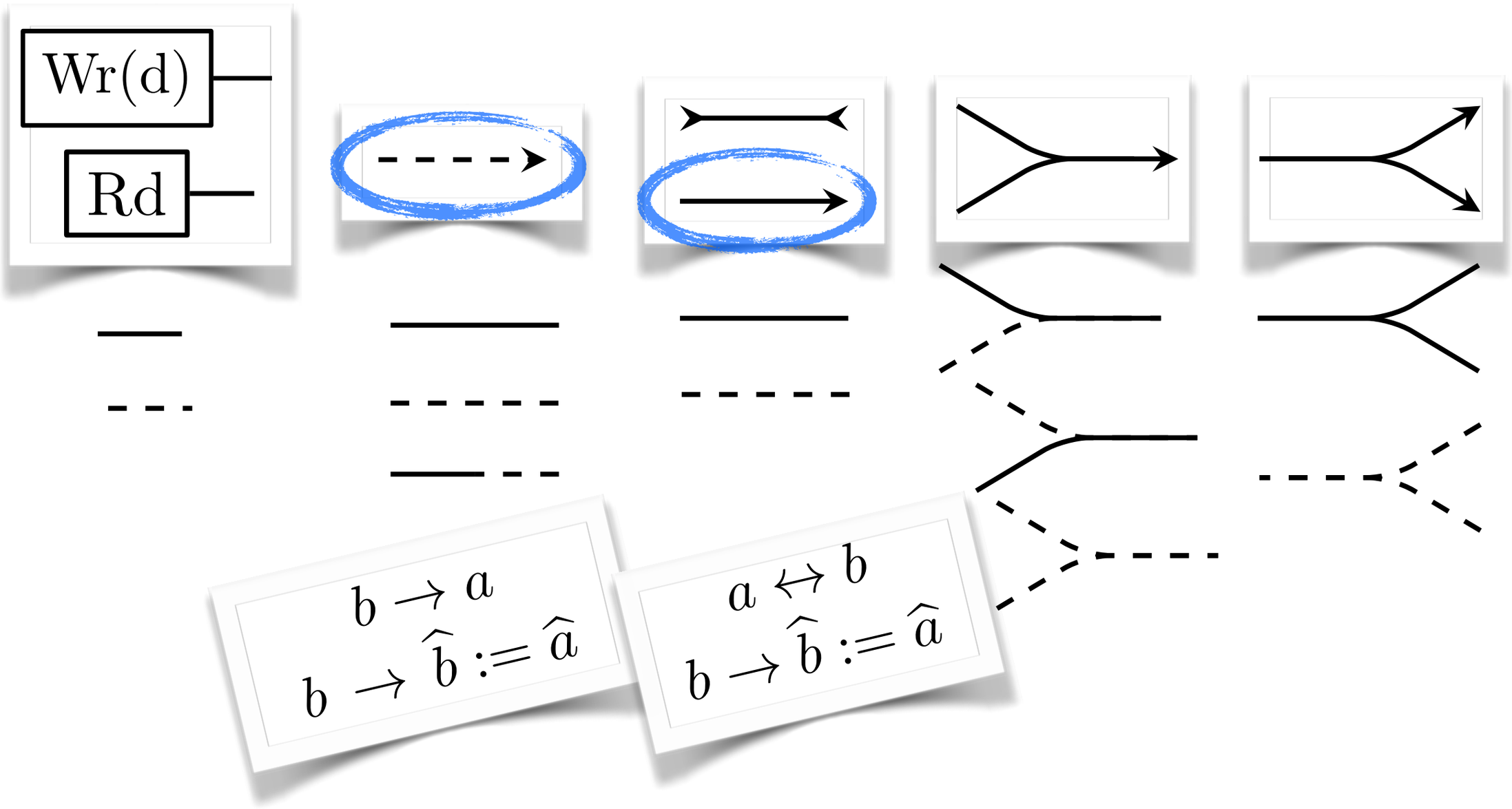
Reo as (Interactive) Constraints

Coordination as constraint satisfaction

$a \longrightarrow b$	$a \leftrightarrow b$ $b \to \widehat{b} := \widehat{a}$
$a \dashrightarrow b$	$b \to a$ $b \to \widehat{b} := \widehat{a}$
$a \overset{P}{\rightsquigarrow} b$	$b \to \widehat{b} := \widehat{a}$ $(a \wedge P(\widehat{a})) \leftrightarrow b$
$a \overset{f}{\rightarrow\!\!\triangleright} b$	$a \leftrightarrow b$ $b \to \widehat{b} := f(\widehat{a})$



Exercise: write constraints



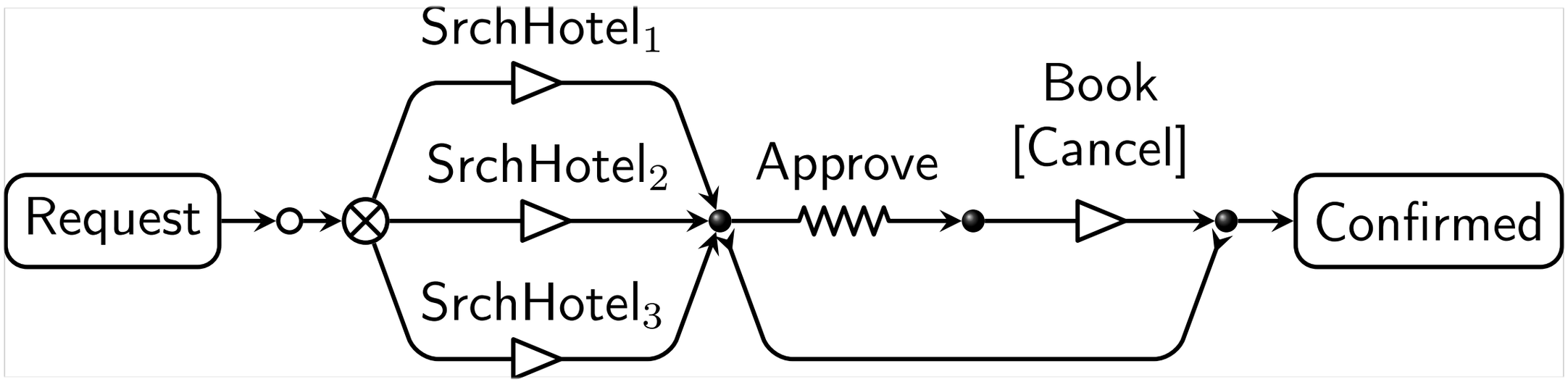
Building up constraints

- Connector colouring as constraints
- Data constraints
- Interactive constraints

Context

(need for extra interaction)

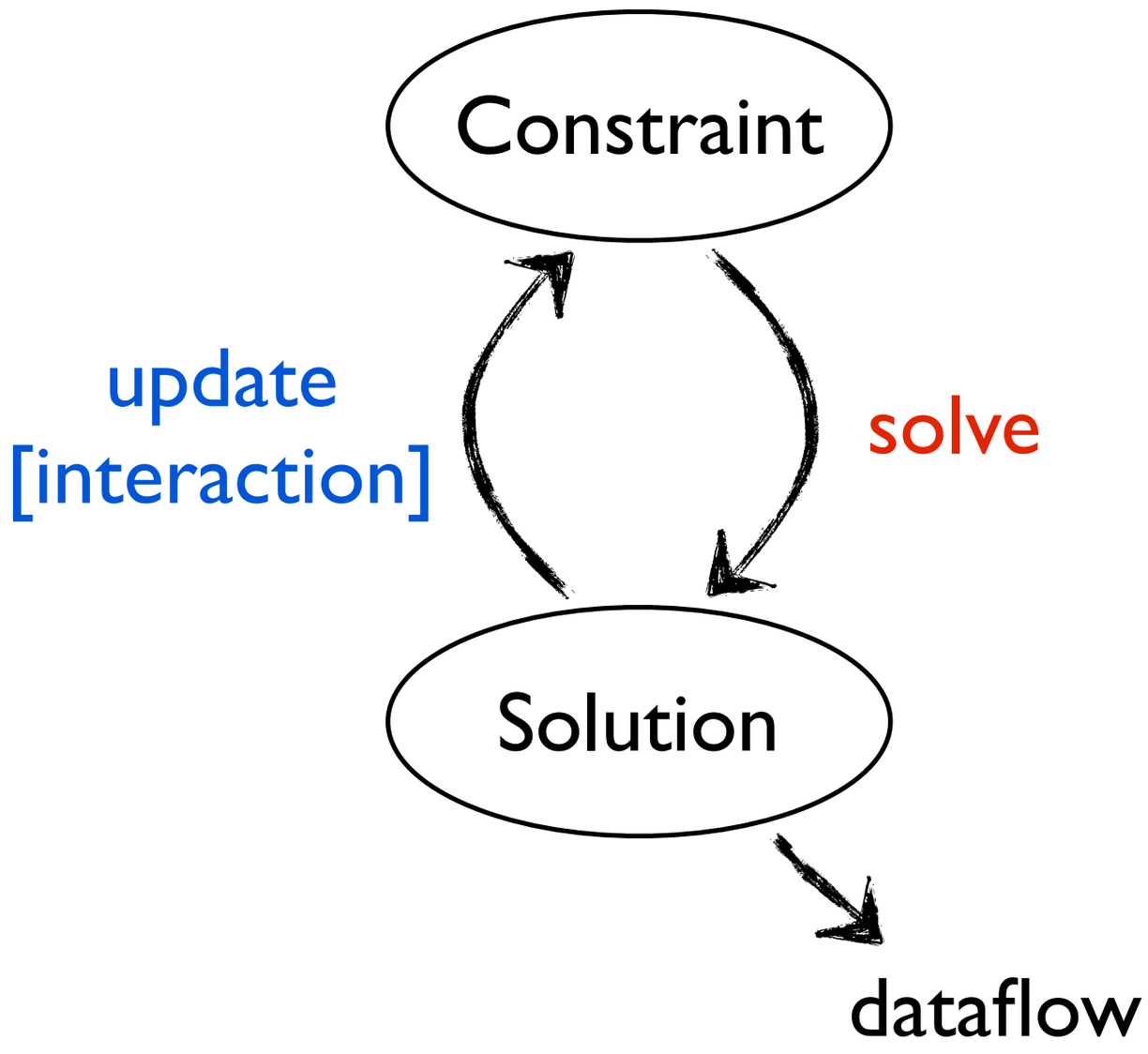
Hotel booking

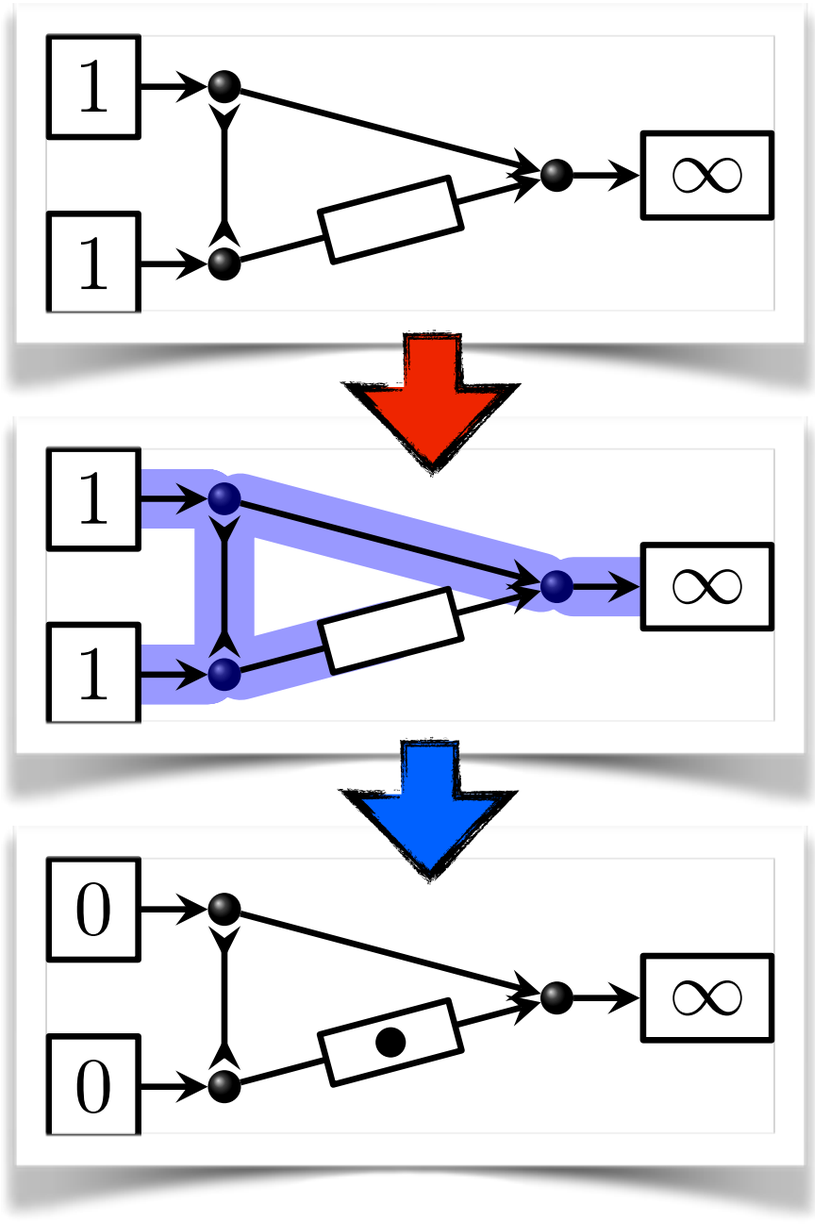


- Interaction with hotel repositories
- Interaction with users
- Interaction with hotels (availability & payment)

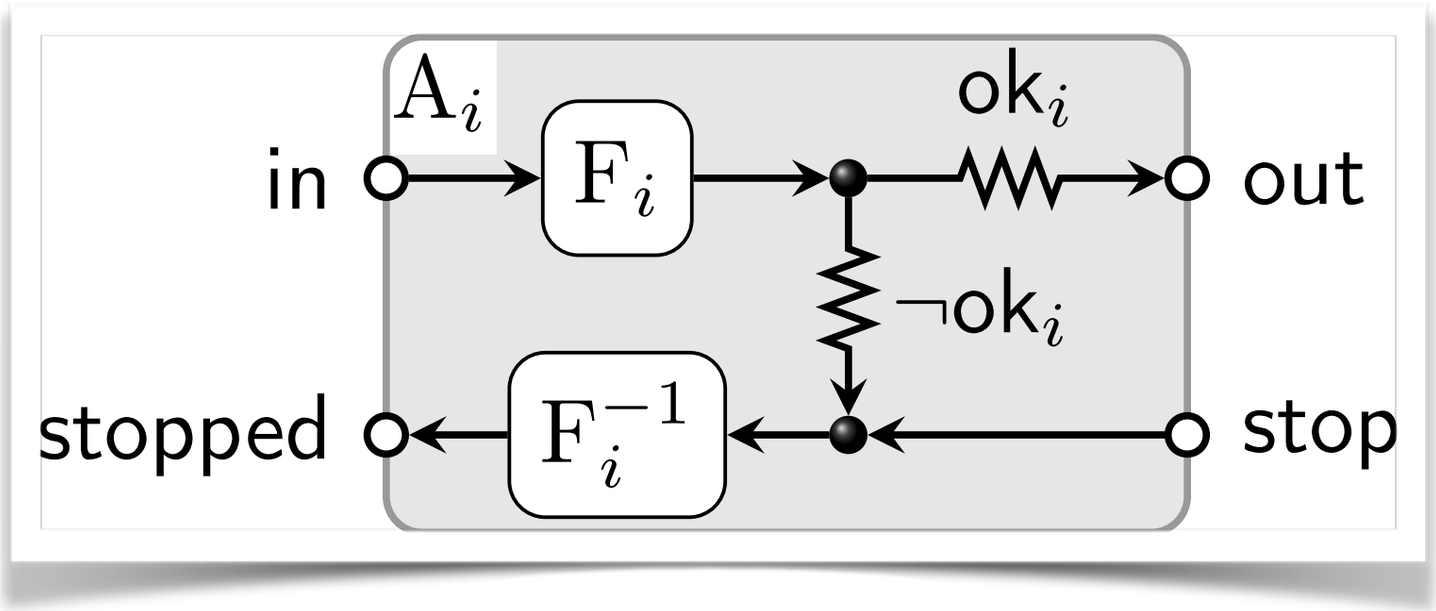
Problem of Interaction

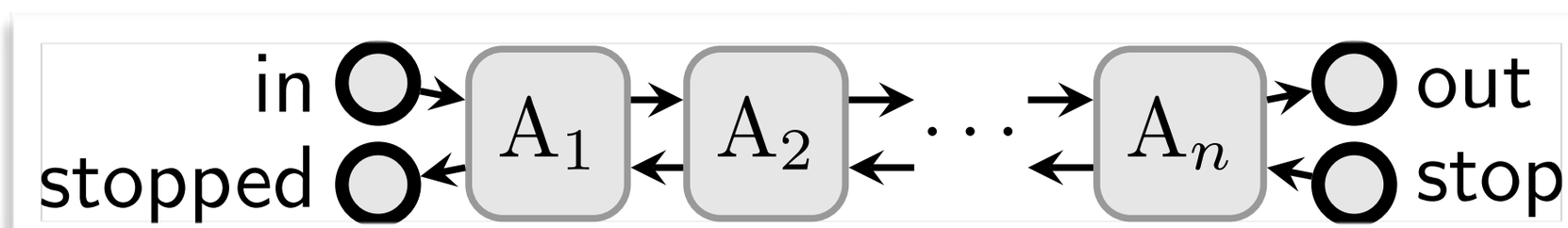
Coordination as constraint satisfaction



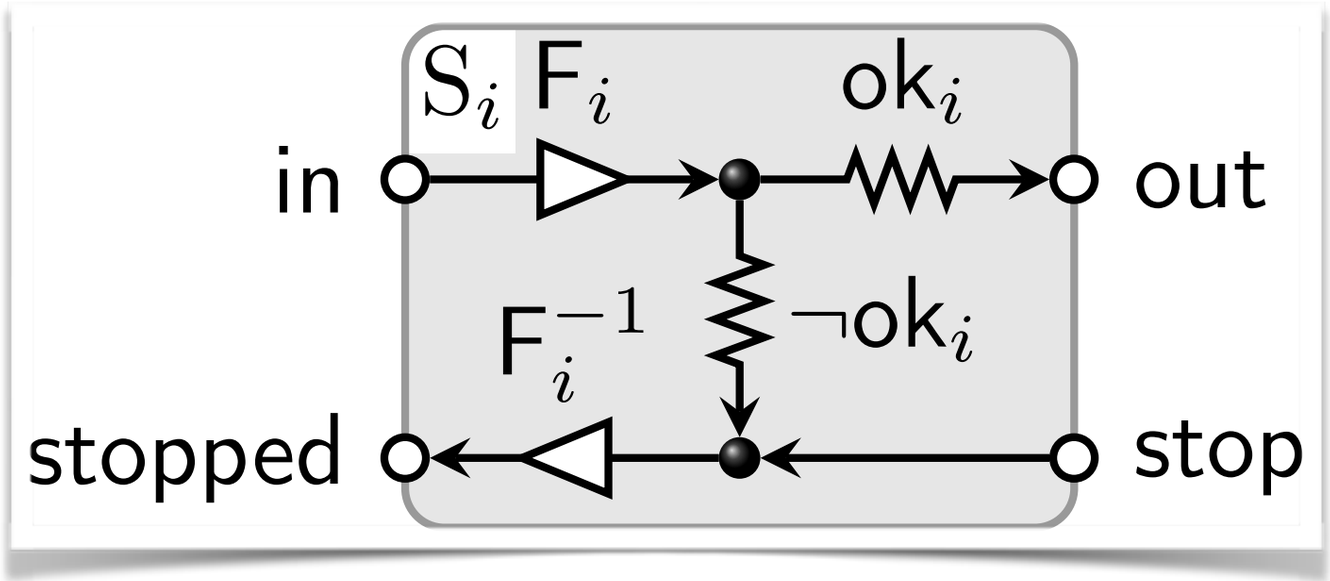


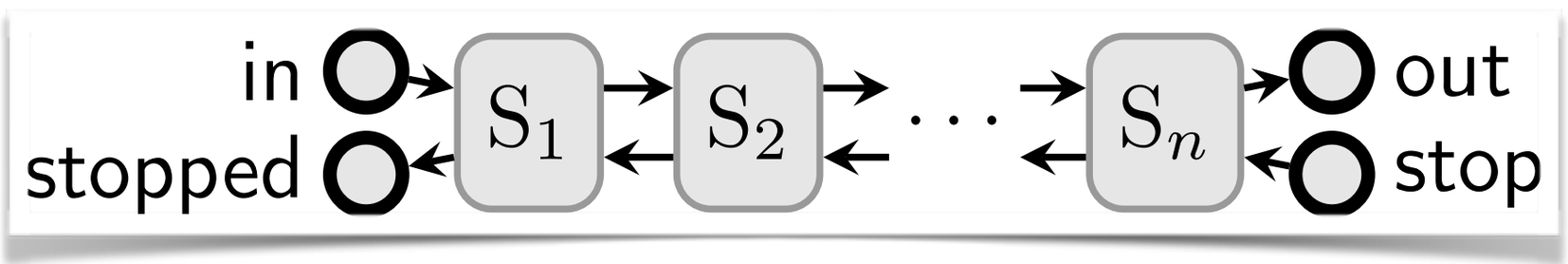
Asynchronous transactions





Synchronous transactions





Problem

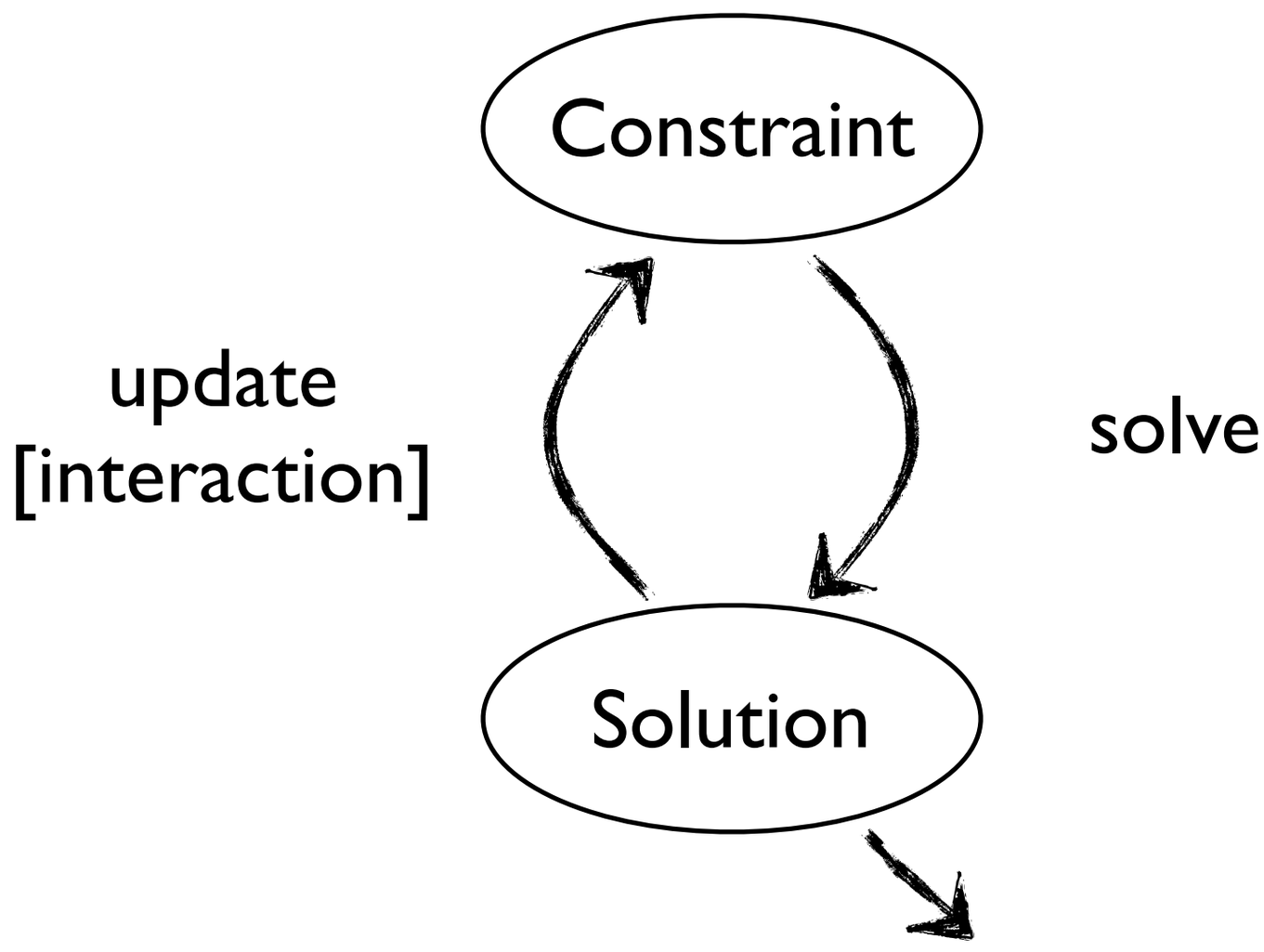
Interaction or Synchrony

but not both

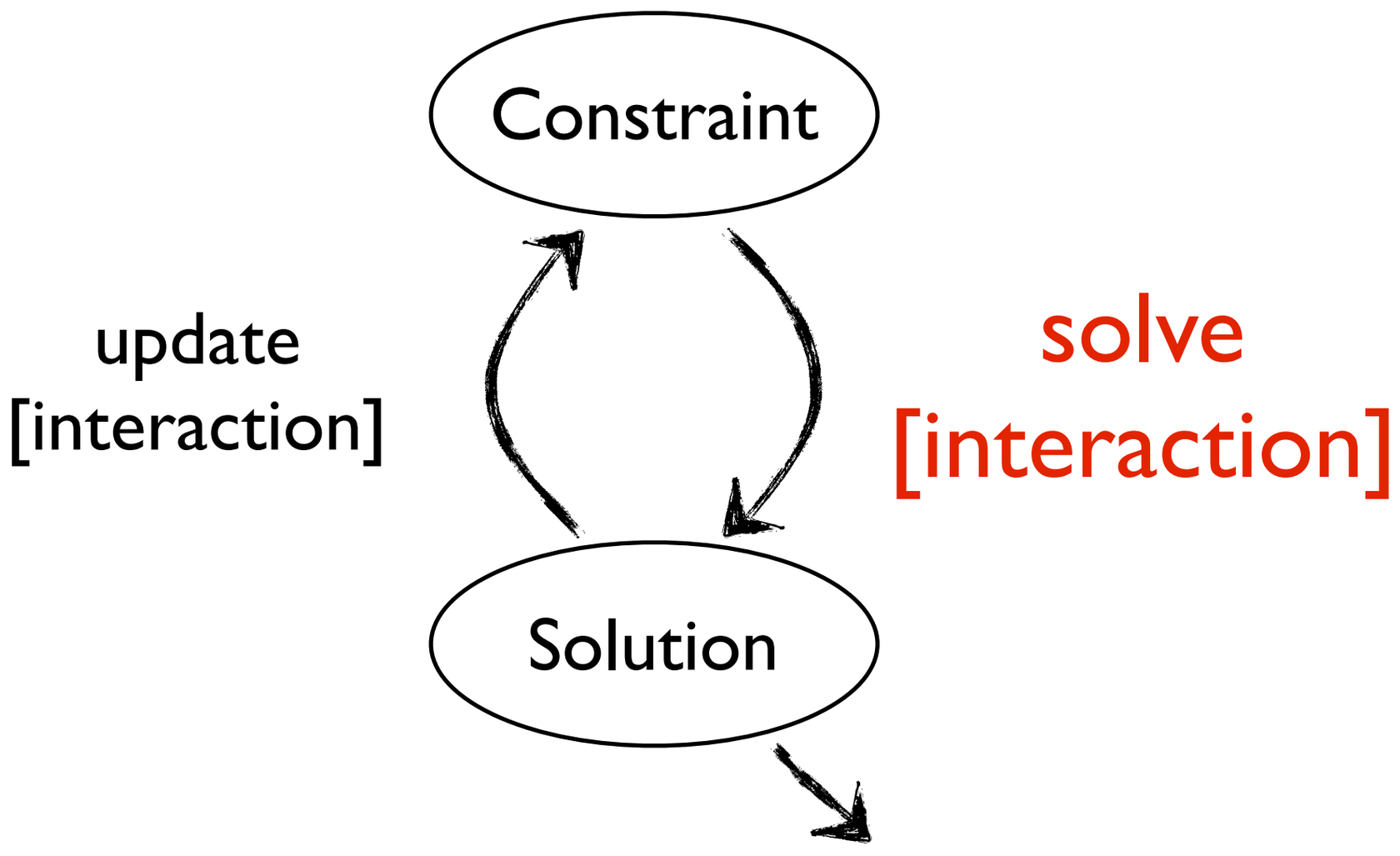
Solution

Interactive Interaction Constraints

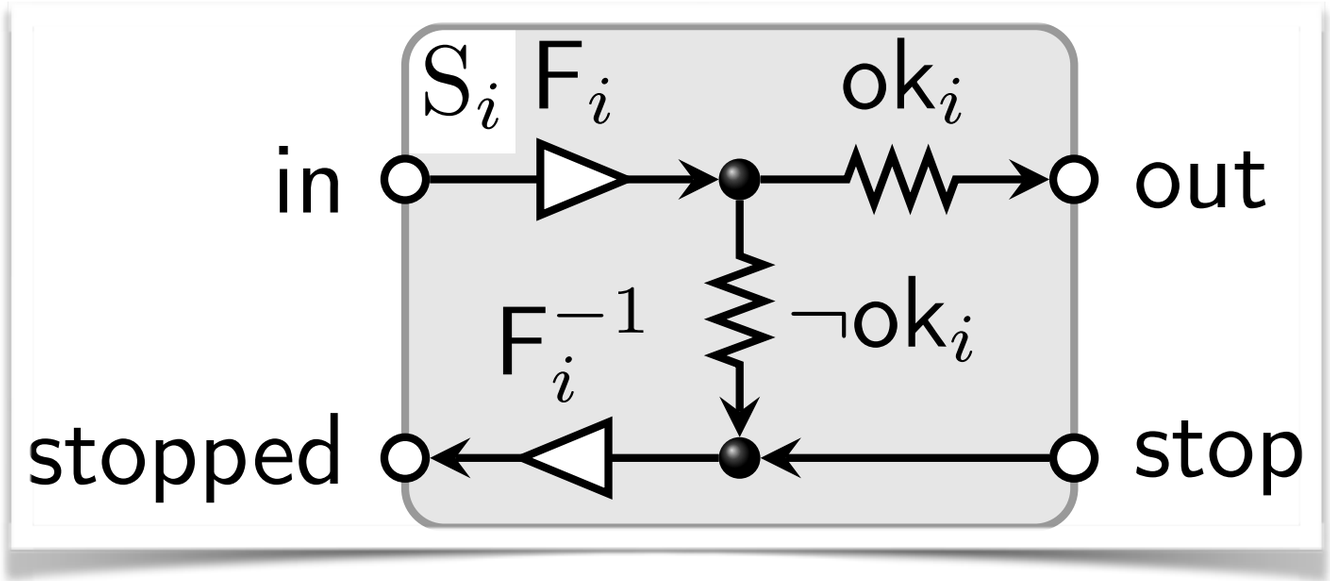
Coordination as constraint satisfaction

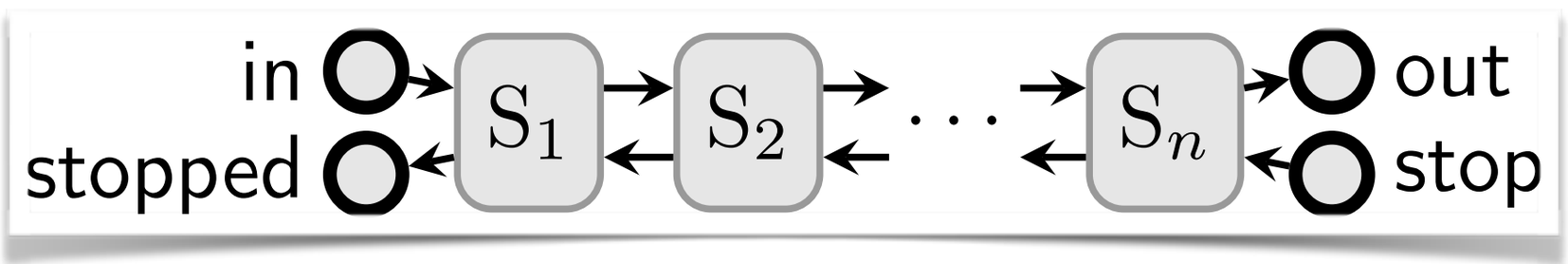


Coordination as interactive constraint satisfaction

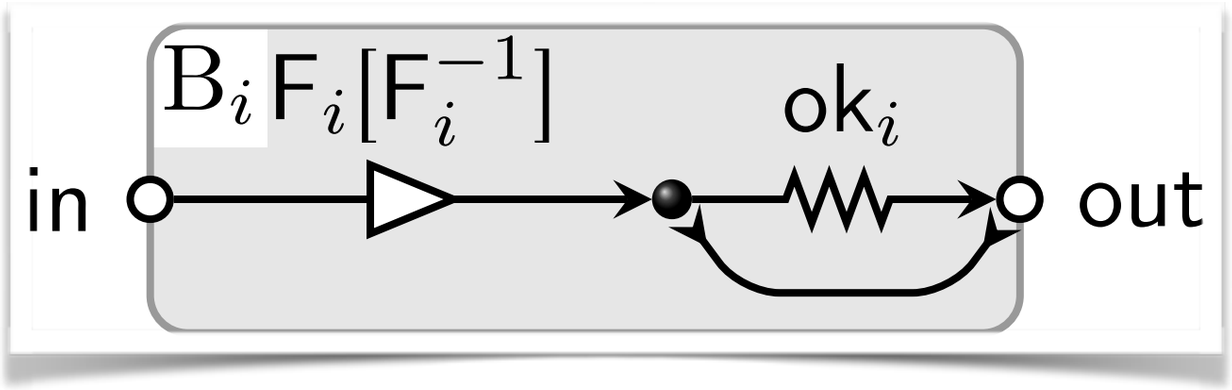


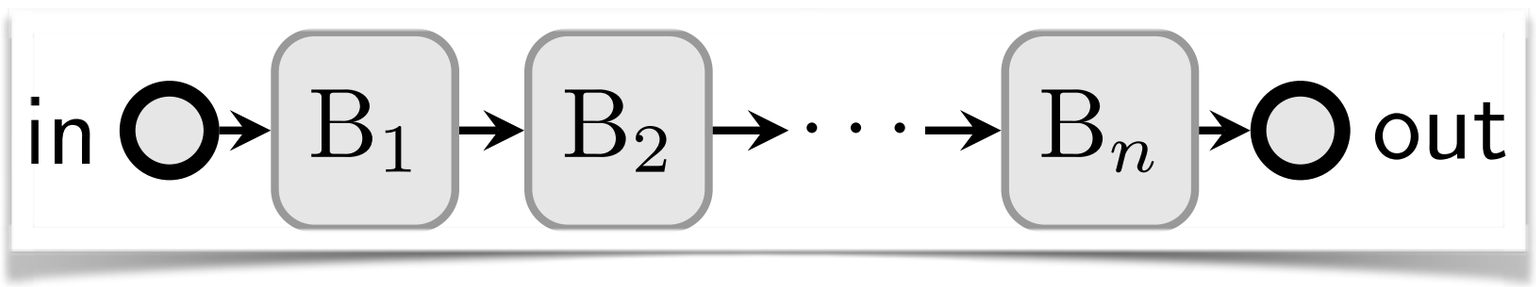
Synchronous transactions



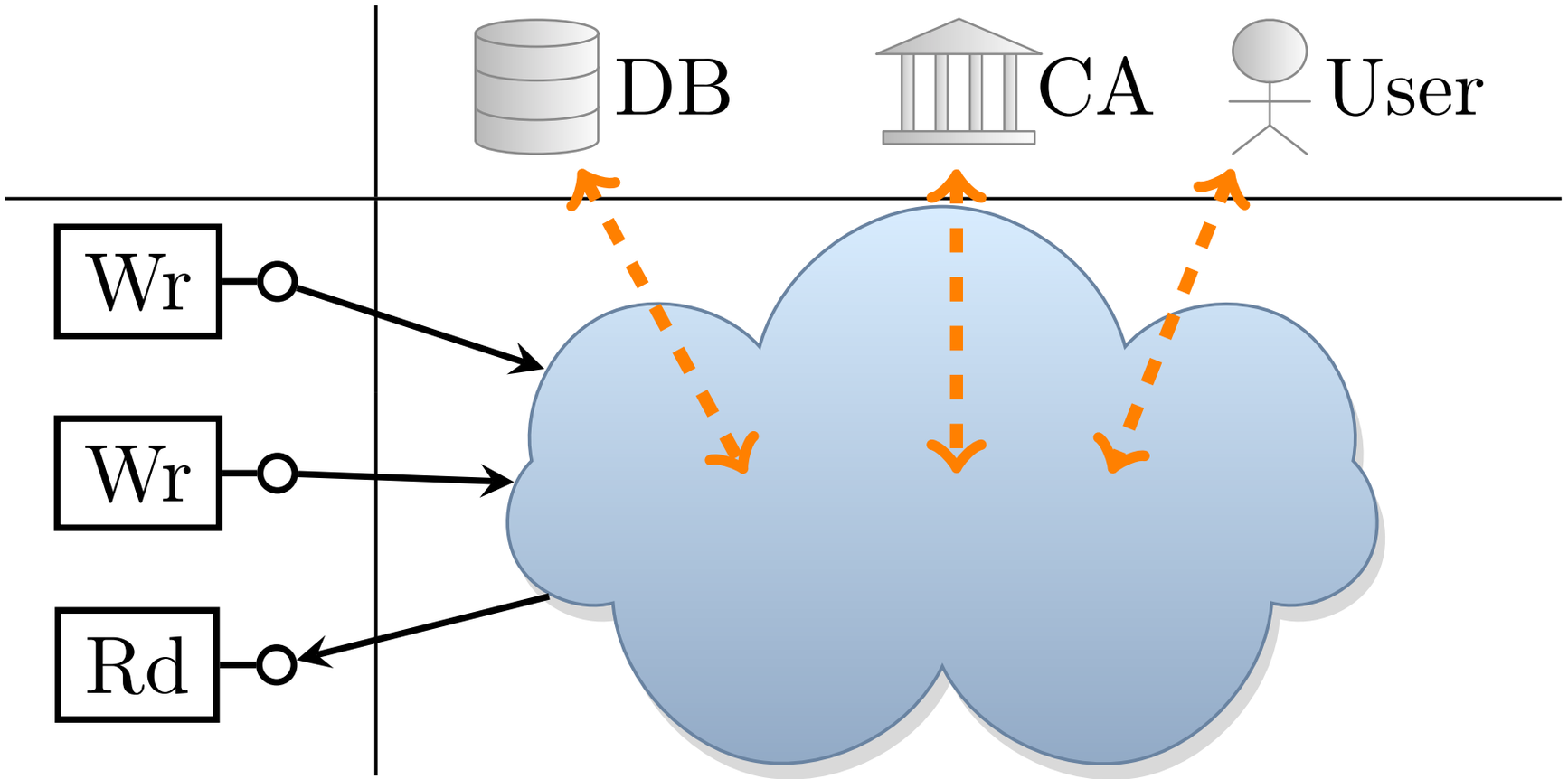


Synchronous transactions

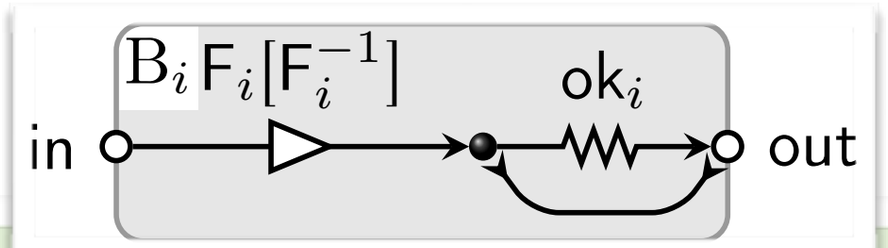




Interactive Interaction Constraints

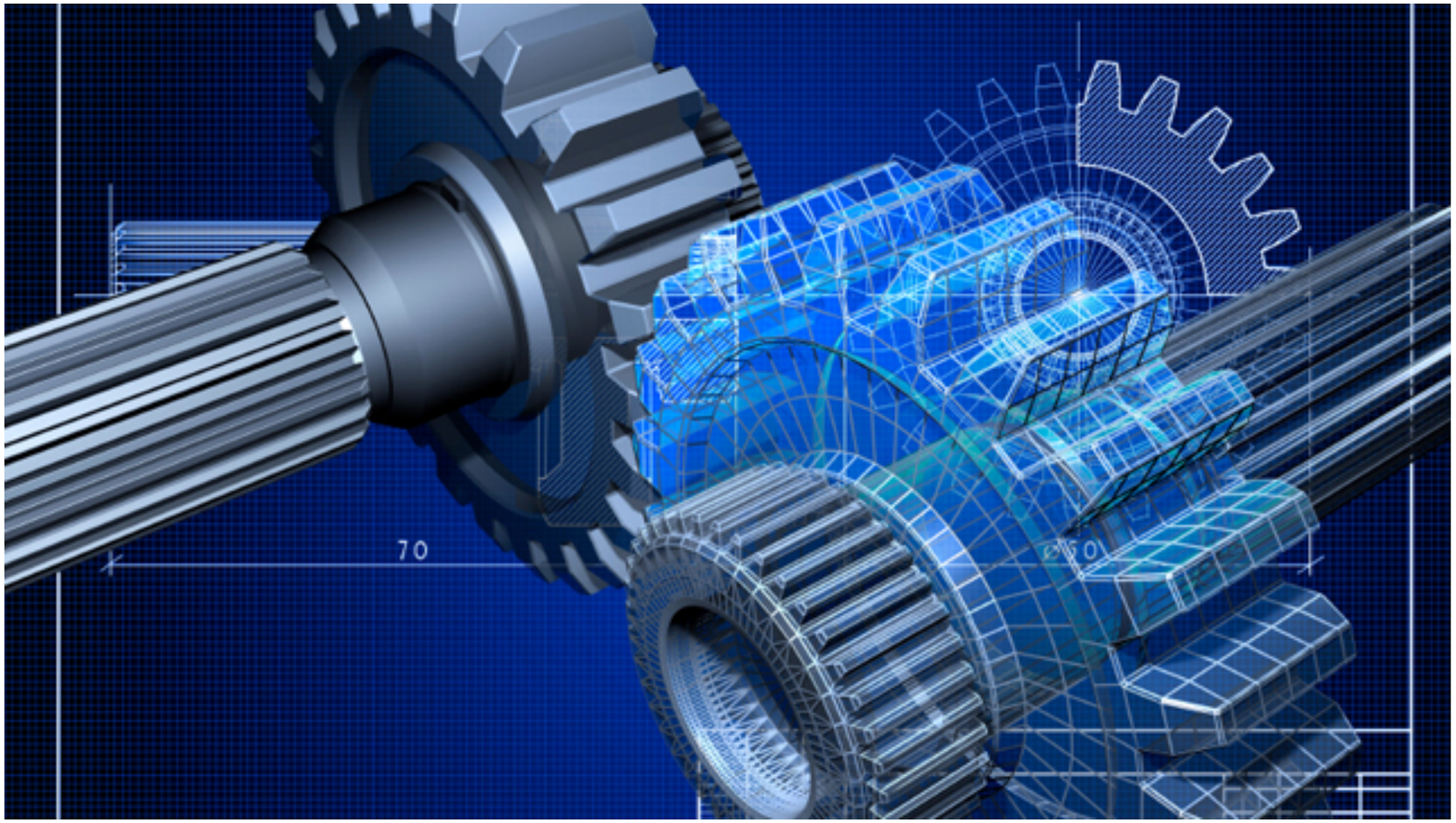


Scala / Java implementation




```
val f = Function("f") {
  case s: String => /* do something */
}

val finv = Function("f^-1") {
  case s: String => /* do something */
}

val ok = Predicate("ok") {
  case s: String => /* do something */
}

val connector =
  writer("in",List("a","b")) ++
  transf("in","x",f,finv)    ++
  filter("x","out",ok)       ++
  sdrain("x","out")          ++
  reader("out",2)

connector.run()
```

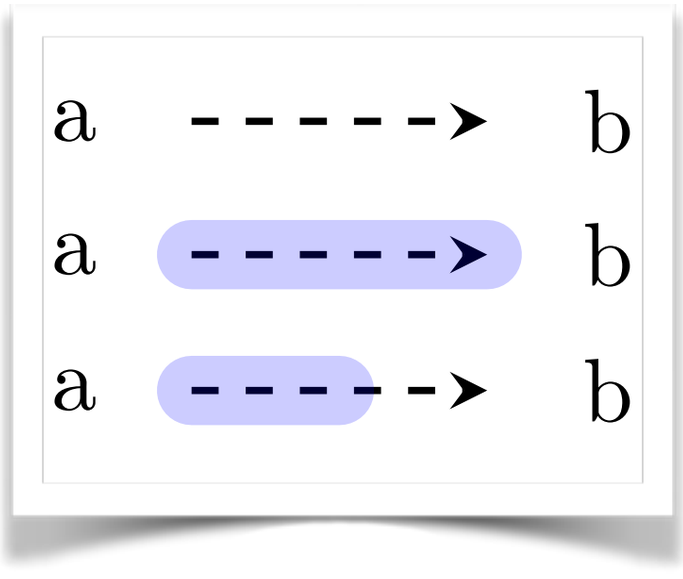
```
class Filter(a: String, b: String,
             p: Predicate)
            extends ... {

  def getConstraint = Constraint(
    b --> a,
    b --> (b := a),
    b --> (a :< p),
    (a /\ (a :< p)) --> b
  )

  /* override def update(s:
       Option[Solution]) = ... */
}
```



More about constraints

CC2 as constraints

- [*Colouring*: End → {Flow, NoFlow}]
- *Formula:* Boolean over End
- *Composition* = conjunction

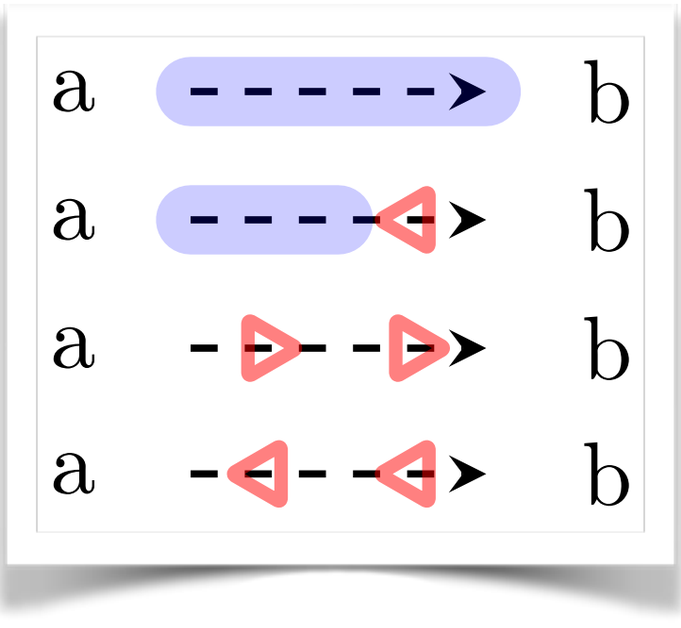


$$(\neg a \wedge \neg b) \quad \vee$$
$$(a \wedge b) \quad \vee$$
$$(a \wedge \neg b)$$

$$b \rightarrow a$$

CC3 as constraints

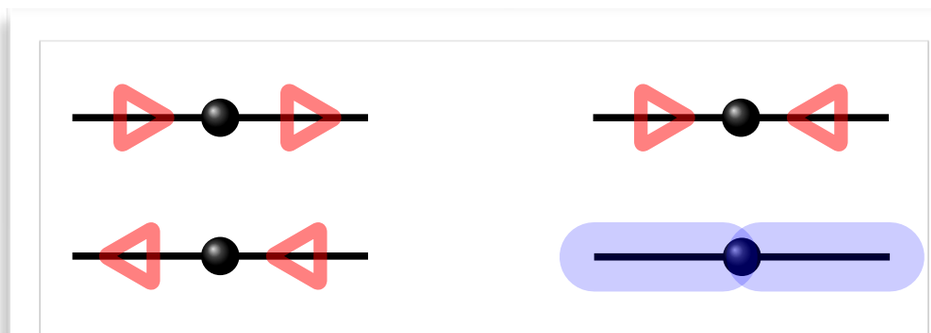
- [*Colouring*: End → {Flow, GiveReason, GetReason}]
- *Formula:* Boolean over End, End_{src}, End_{snk}
- a = flow on a ; a_{src} = give reason ; $\neg b_{snk}$ = get reason



$$b \rightarrow a \wedge$$
$$(a \wedge \neg b) \rightarrow (a \wedge \neg b_{snk}) \wedge$$
$$\neg a \rightarrow (\neg b \wedge \neg a_{src} \wedge b_{snk})$$

CC3 as constraints
Composition

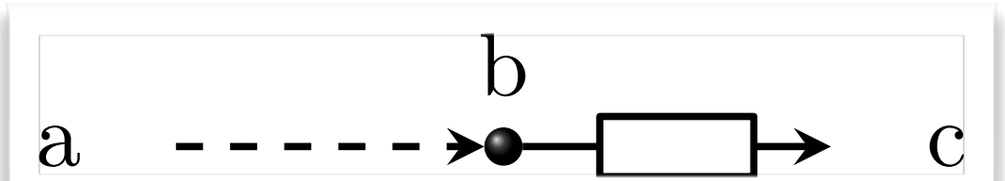
- a = flow on a ; a_{src} = give reason ; $\neg b_{snk}$ = get reason
- one-to-one composition: source to sink ends



$$\forall x \quad \cdot \quad x_{snk} \vee x_{src}$$

CC3 as constraints

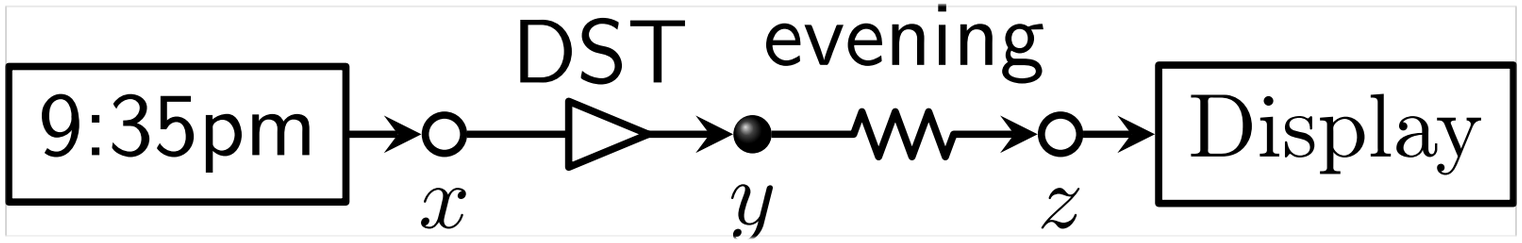
example



$$
\begin{aligned}
\phi \quad = \quad & b \rightarrow a \wedge \neg c \\
& (a \wedge \neg b) \rightarrow (a \wedge \neg b_{snk}) \wedge \\
& \neg a \rightarrow (\neg b \wedge \neg a_{src} \wedge b_{snk}) \wedge \\
& (\neg b \rightarrow \neg b_{src}) \wedge c_{snk} \wedge \\
& b_{src} \vee b_{snk}
\end{aligned}
$$

$$
\begin{aligned}
\{a \wedge b \wedge \neg c \wedge c_{snk}\} \quad & \models \quad \phi \\
\{\neg a \wedge \neg b \wedge \neg c \wedge \neg a_{src} \wedge b_{snk} \wedge \neg b_{src} \wedge c_{snk}\} \quad & \models \quad \phi
\end{aligned}
$$

Data constraints

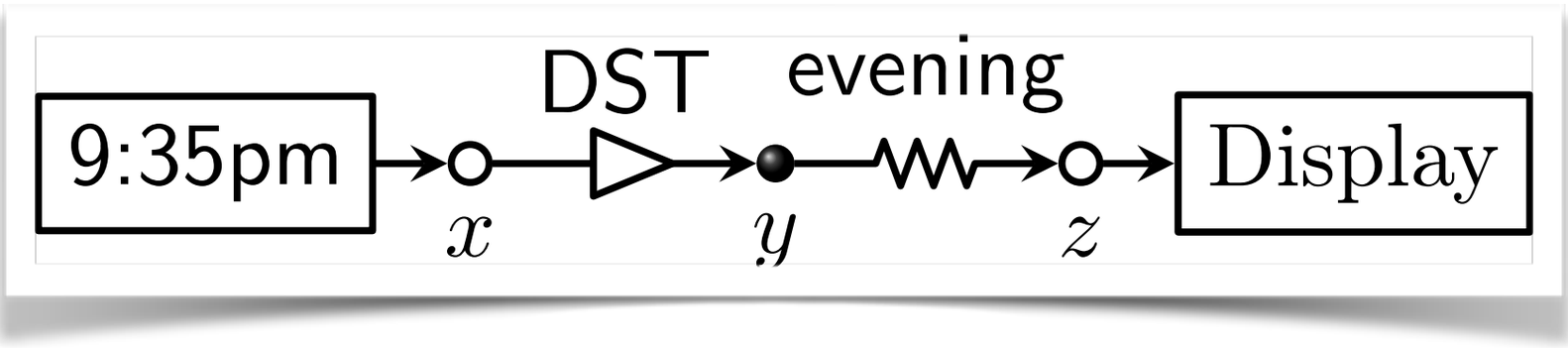


$x \to \widehat{x} :=$ 9:35pm $\quad x \leftrightarrow y \quad y \to \widehat{y} :=$ DST$(\widehat{x})$

$(y \wedge$ evening$(\widehat{y})) \leftrightarrow z \quad z \to \widehat{z} := \widehat{y}$

How to solve this?

Predicate abstraction



original

$$x \to \widehat{x} := \text{9:35pm} \qquad x \leftrightarrow y \qquad y \to \widehat{y} := \text{DST}(\widehat{x})$$

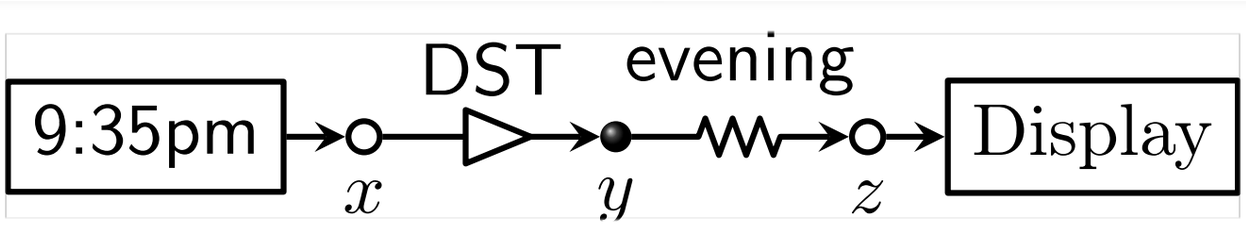
$$(y \wedge \text{evening}(\widehat{y})) \leftrightarrow z \qquad z \to \widehat{z} := \widehat{y}$$

boolean

$$x \to \widehat{x}_{\text{ev.dst}} := [\text{evening}(\text{DST}(\text{9:35pm}))] \qquad x \leftrightarrow y$$

$$y \to \widehat{y}_{\text{ev}} := \widehat{x}_{\text{ev.dst}} \qquad (y \wedge \widehat{y}_{\text{ev}}) \leftrightarrow z$$

Interaction via Choco



boolean

$$x \to \widehat{x}_{\mathsf{ev.dst}} := [\mathsf{evening}(\mathsf{DST}(\mathsf{9{:}35pm}))] \qquad x \leftrightarrow y$$

$$y \to \widehat{y}_{\mathsf{ev}} := \widehat{x}_{\mathsf{ev.dst}} \qquad (y \wedge \widehat{y}_{\mathsf{ev}}) \leftrightarrow z$$

interactive

$$x \to \mathsf{XPred}(\mathsf{ev.dst}, x, \mathsf{9{:}35pm}) \qquad x \leftrightarrow y$$

$$y \to \widehat{y}_{\mathsf{ev}} := \widehat{x}_{\mathsf{ev.dst}} \qquad (y \wedge \widehat{y}_{\mathsf{ev}}) \leftrightarrow z$$

Interaction via Choco

- instance of a Choco constraint
- reacts when x or $\widehat{x}_{\text{ev.dst}}$ is instantiated
- $\neg x \Rightarrow \widehat{x}_{\text{ev.dst}}$ can be anything
- $x \Rightarrow \widehat{x}_{\text{ev.dst}} = \text{ev}(\text{dst}(9{:}35\text{pm}))$

9:35pm

boolean

$x \to \widehat{x}_{\text{ev.dst}} = [\text{evening}(\text{DST}(9{:}35\text{pm}))] \qquad x \leftrightarrow y$

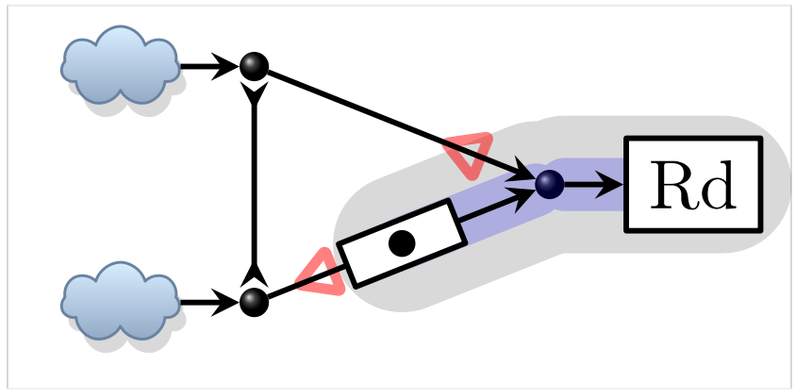
$y \to \widehat{y}_{\text{ev}} := \widehat{x}_{\text{ev.dst}} \qquad (y \wedge \widehat{y}_{\text{ev}}) \leftrightarrow z$

interactive

$x \to \text{XPred}(\text{ev.dst}, x, 9{:}35\text{pm}) \qquad x \leftrightarrow y$

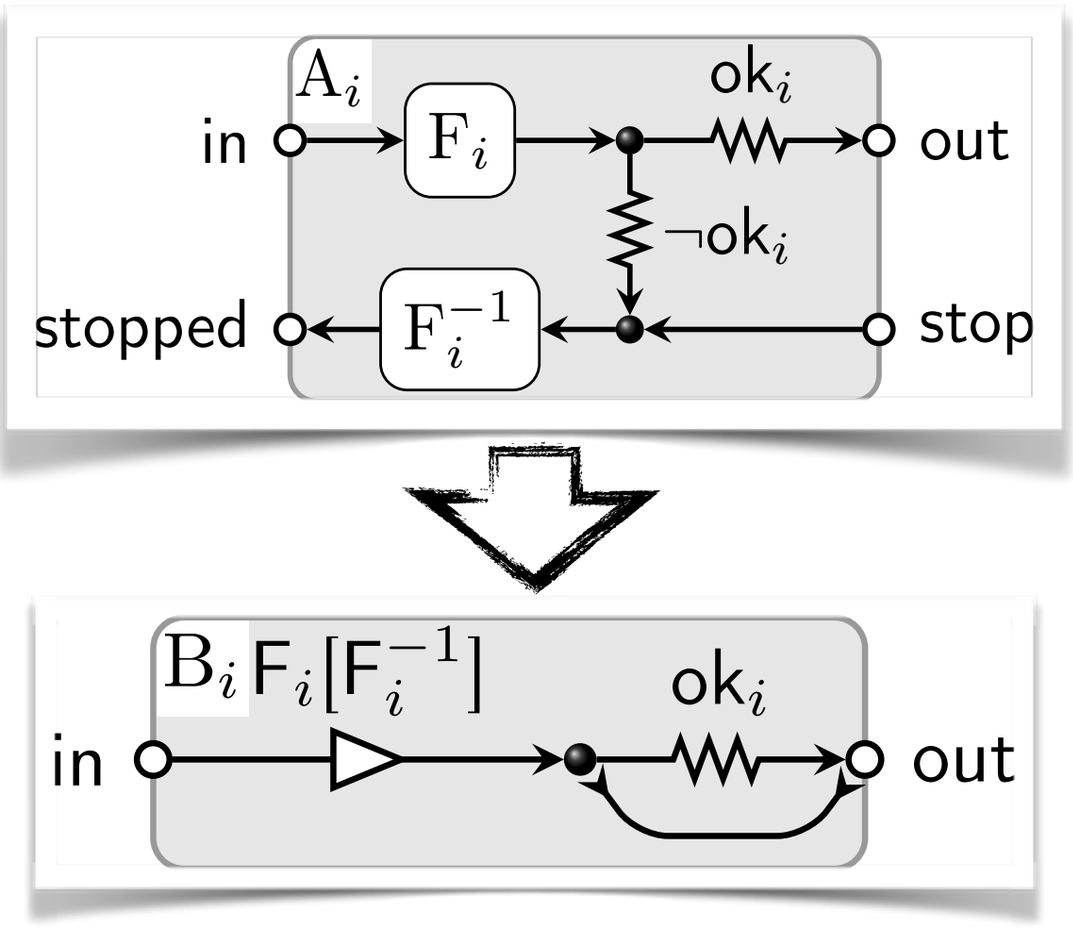
$y \to \widehat{y}_{\text{ev}} := \widehat{x}_{\text{ev.dst}} \qquad (y \wedge \widehat{y}_{\text{ev}}) \leftrightarrow z$

Wrapping up



- Interactive constraint solving
- Expose the atomicity of Reo to components

- Reo and connector colouring



Ongoing experiments

- Avoiding pre-processing (SMT instead of SAT) $\exists a, \hat{a} \cdot \psi$
- Compiling steps:
- Heuristics for identifying potential partial colourings
- Combining local and interactive constraints (probably to a journal)

```
object HotelReservation extends App {

  case class Req(val content:String)

  def srchHotel(i:Int) =
        Function("SearchHotel-"+i){
    case r:Req ⇒ i match {
      case 1 ⇒ List("F1","Ibis","Mercury")
      case 2 ⇒ List("B&B","YHostel")
      case _ ⇒ List("HotelA","HotelB")
    }
  }
  val approve = Predicate("approve"){
    case l:List[String] ⇒
      println("approve: "+l.mkString(",
            ")+". [y,n]")
      readChar() == 'y'
  }
  val book = Function("book"){
    case l : List[String] ⇒
      println("Options: "+l.mkString(", ")+
        ". Which one? (1.."+l.length+")")
      val res = readInt()
      l(res-1)
  }
  val cancelB = Function("cancelB"){
    case x ⇒ println("canceling "+x+".")
  }
  val invoice = Function("invoice"){
    case x ⇒ println("invoice for "+x+".")
  }
```

```
  val pay = Predicate("paid"){
    case x ⇒ if (x == "Ibis") {
      println("paid for Ibis")
      true
    }
    else {
      println("not paid for "+x)
      false
    }
  }



  // Connector definition
  val connector =
    writer("req",List(Req("req1"),
          Req("req2"))) ++
    nexrouter("req",List("h1","h2","h3")) ++
    transf("h1","h1o",srchHotel(1)) ++
    transf("h2","h2o",srchHotel(2)) ++
    transf("h3","h3o",srchHotel(3)) ++
    nmerger(List("h1o","h2o","h3o"),"hs") ++
    filter("hs","ap",approve) ++
    sdrain("hs","ap") ++
    transf("ap","bk",book,cancelB) ++
    monitor("bk","inv",invoice) ++
    filter("inv","paid",paid) ++
    reader("paid",5)

  connector.run()
}
```