Composition and Refinement of Behavioral Specifications

Dusko Pavlovic
Douglas R. Smith

Kestrel Institute
Palo Alto, California

www.kestrel.edu

Specifications and Morphisms

spec **Partial-Order** is
 sort E
 op le: E, E $\rightarrow$ Boolean
 axiom reflx is le(x,x)
 axiom trans is le(x,y) $\wedge$ le(y,z) $\Rightarrow$ le(x,z)
 axiom antis is le(x,y) $\wedge$ le(y,x) $\Rightarrow$ x = y
end-spec

$\longrightarrow$

E $\mapsto$ Int
le $\mapsto$ $\leq$
axioms $\mapsto$ thms

spec **Integer** is
 sort Int
 op $\leq$: Int, Int $\rightarrow$ Boolean
 op 0 : Int
 op _+_ : Int , Int $\rightarrow$ Int
 …
end-spec

Specification morphism: a language translation that preserves provability

Specification Carrying Software

$$\langle P, \models, S \rangle$$

P	$=$	program
S	$=$	specification
$\models$	$=$	model relation

Specification Carrying Software

$$A = \langle P_A, \models_A, S_A \rangle$$

$$f \downarrow \qquad f_P \uparrow \qquad f_S \downarrow$$

$$B = \langle P_B, \models_B, S_B \rangle$$

$$f_P(b) \models_A \alpha$$

$$\Updownarrow$$

$$b \models_B f_S(\alpha)$$

simulates behavior

interprets structure

Evolving specifications (especs)

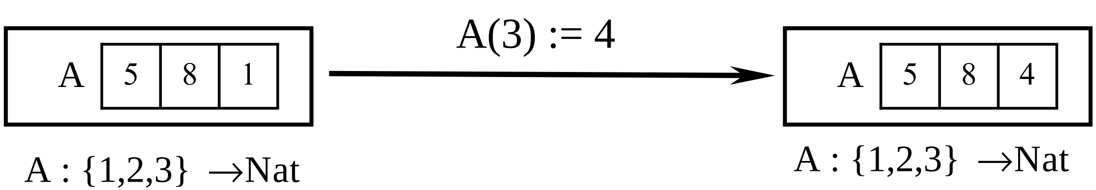
Key ideas that link state machine concepts with logical concepts

1. States are models (structures satisfying axioms)

State	Model
datatypes	sets
variables	functions, values
properties	axioms, theorems

2. State transitions are finite model changes

Example: Updating an array/finite-function A

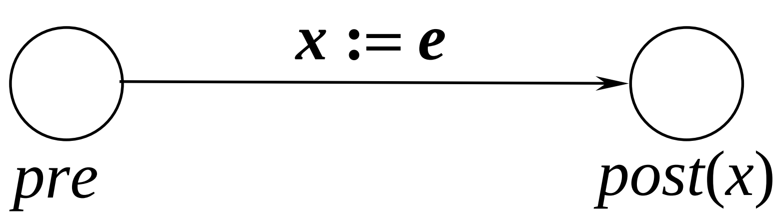


Evolving specifications (especs)

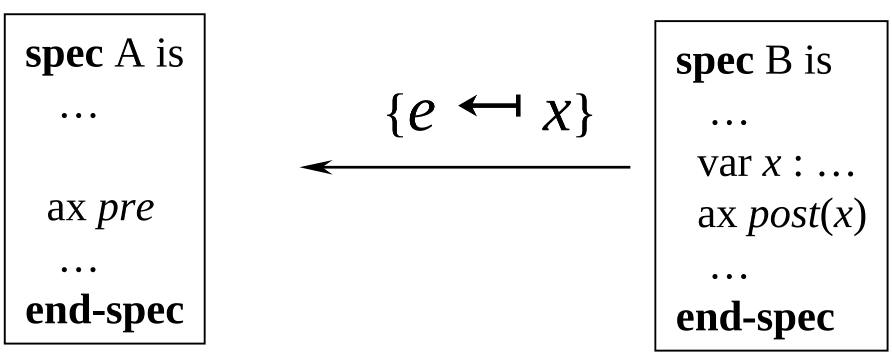
3. Abstract states are sets of states
   Specs denote sets of models

   Specs represent abstract states

4. Abstract transitions are interpretations (in the opposite direction)!

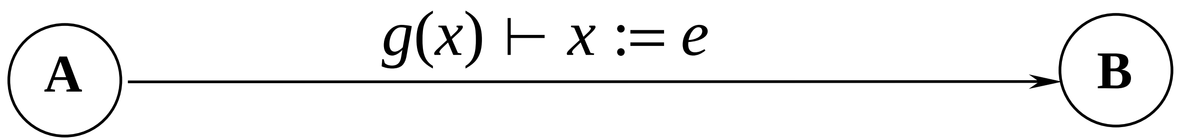


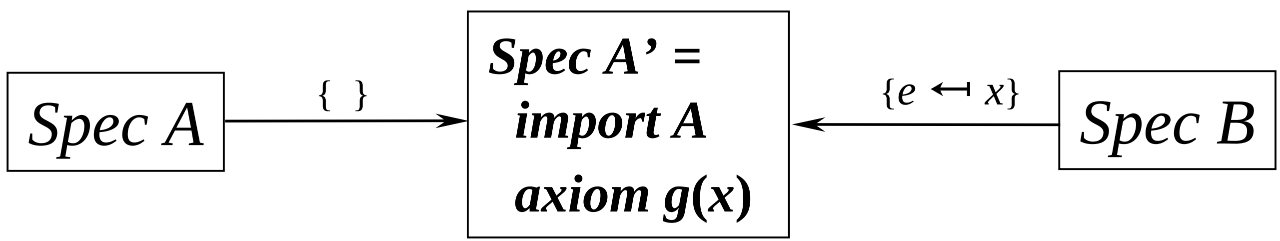
correctness condition
$pre \vdash post(e)$



Evolving specifications (especs)

5. Abstract Guarded Transitions





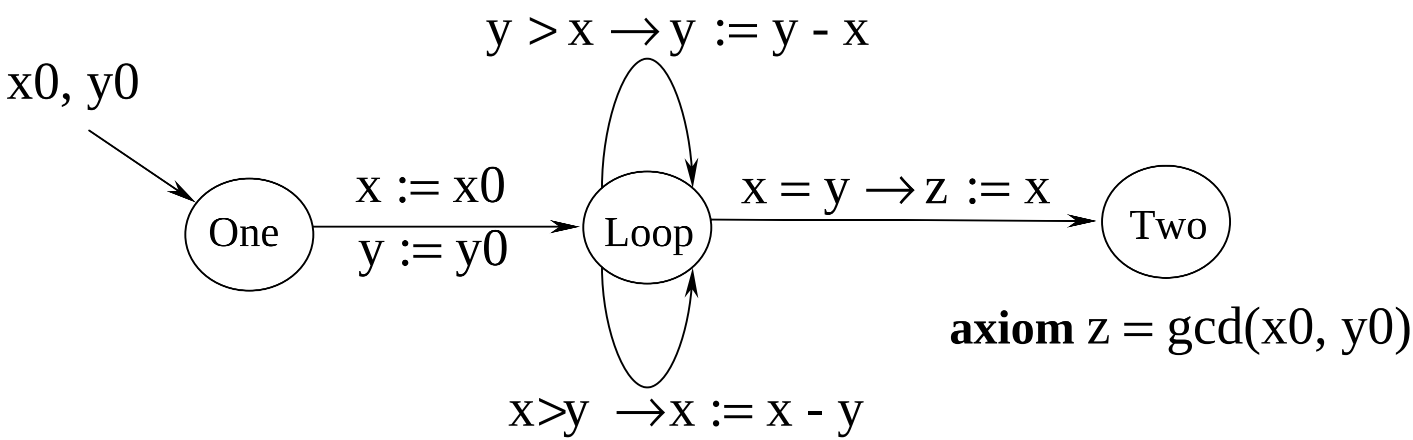
These pairs of arrows are monics and epis of an abstract factorization system, with a general construction for composition and colimit. (AMAST02)

Espec for a GCD Algorithm

Each state has common structure:

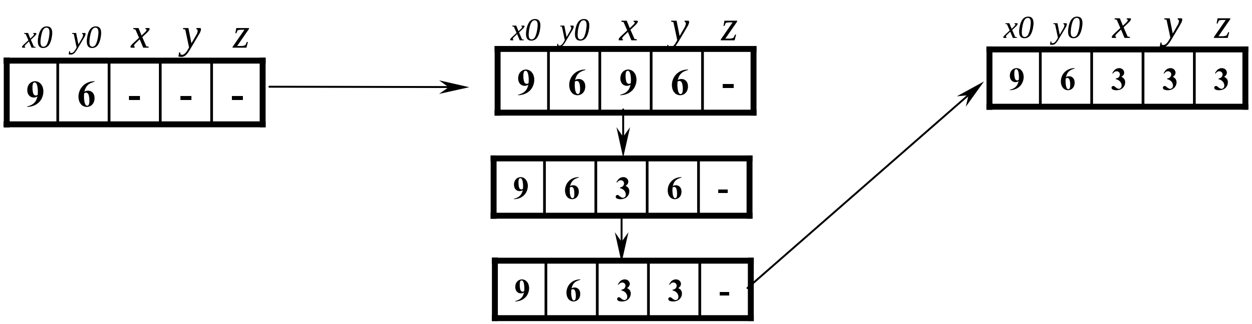
x0, y0 : Pos

x, y, z : Pos



axiom gcd(x0, y0) = gcd(x, y)

example run:



GCD espec

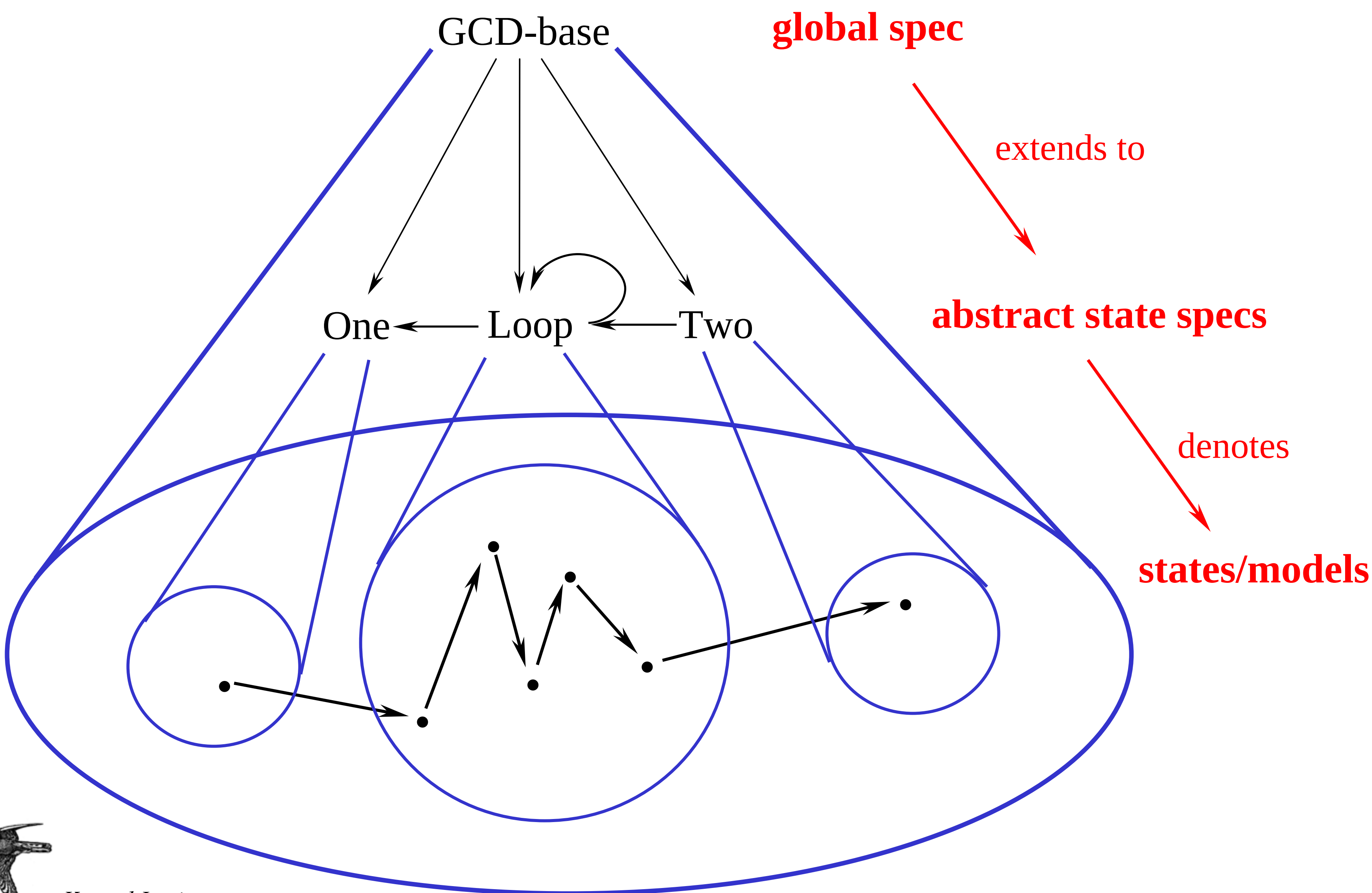
lobal
spec

```
espec GCD-base is
 spec
  op X-in,Y-in : Pos
  op X,Y : Pos
  op Z : Pos
  op gcd : Pos, Pos -> Pos
  axiom gcd-spec is
         gcd(x,y) = z => (divides(z,x) & divides(z,y)
                  & forall(w:Pos)(divides(w,x) & divides(w,y) => w <= z))
 end-spec

 prog
   stad One init[X-in,Y-in] is
   end-stad

   stad Loop is
      thm loop-invariant is
         gcd(X-in,Y-in) = gcd(X,Y)
   end-stad

   stad Two fin[Z] is
      axiom Z = gcd(X-in,Y-in
      axiom X = Y
   end-stad

   step initialize : One -> Loop is
      X := X-in
      Y := Y-in
    end-step

   step Loop1 : Loop -> Loop is
      X := X - Y
      cond X>Y
    end-step

   step Loop2 : Loop -> Loop is
      Y := Y - X
      cond Y>X
    end-step

   step Out : Loop -> Two is
      Z := X
      cond X = Y
    end-step
 end-prog
end-espec
```

ogram
spec

Kestrel Institute

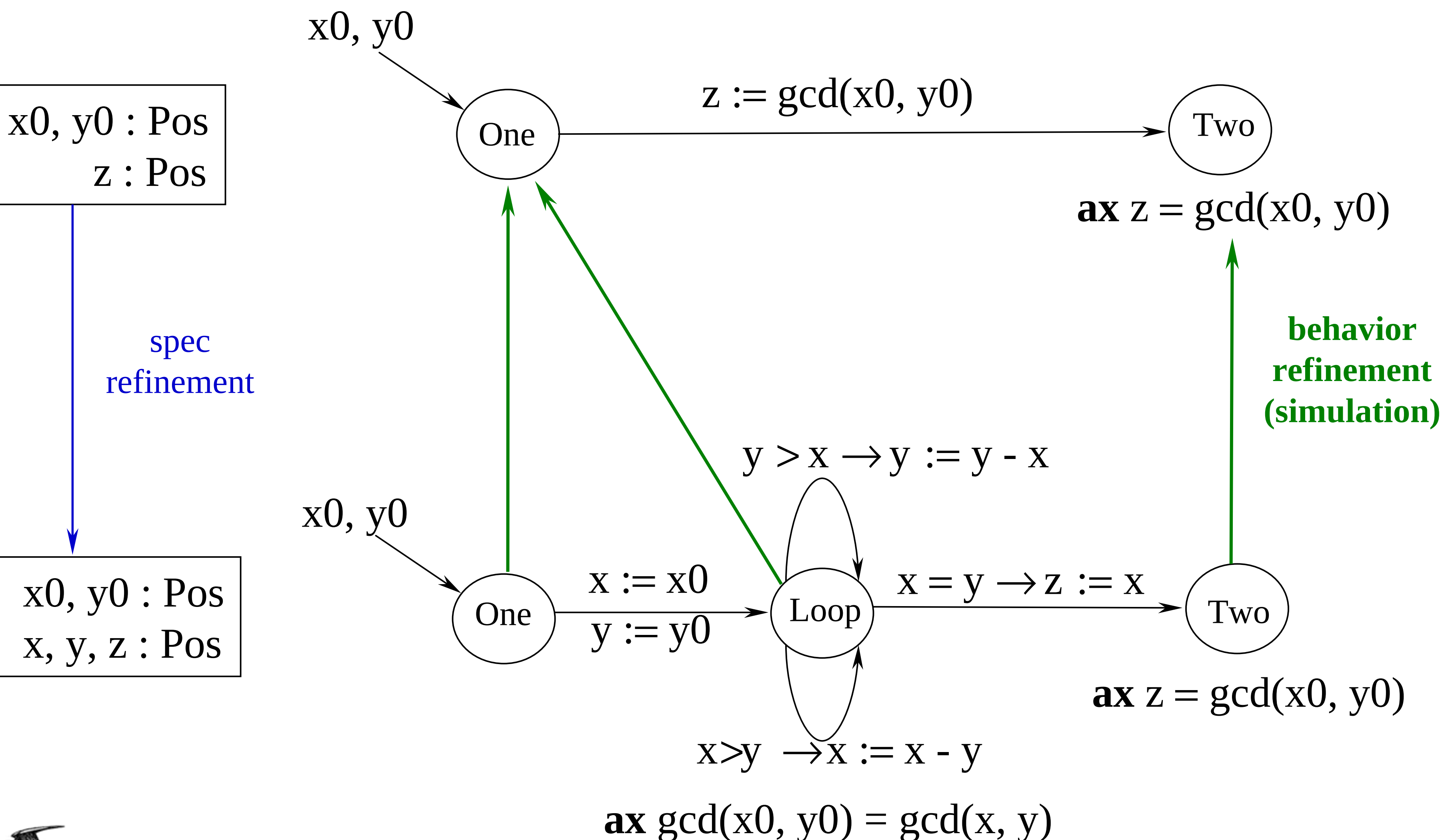
GCD especs, states, and computation



Control Constructs vs Logical Concepts

Command Language	Logical Concepts
$\{P\}\ x := e\ \{Q\}$	interpretation I: $Thy_Q \rightarrow Thy_P$
skip	identity interpretation
sequencing S1;S2	composition $I_1 \circ I_2$
guarded command $g \rightarrow S$	conditional interpretation
if … fi	conditional interpretations with a common codomain
do … od	conditional interpretations with common domain and codomain

Espec Refinement

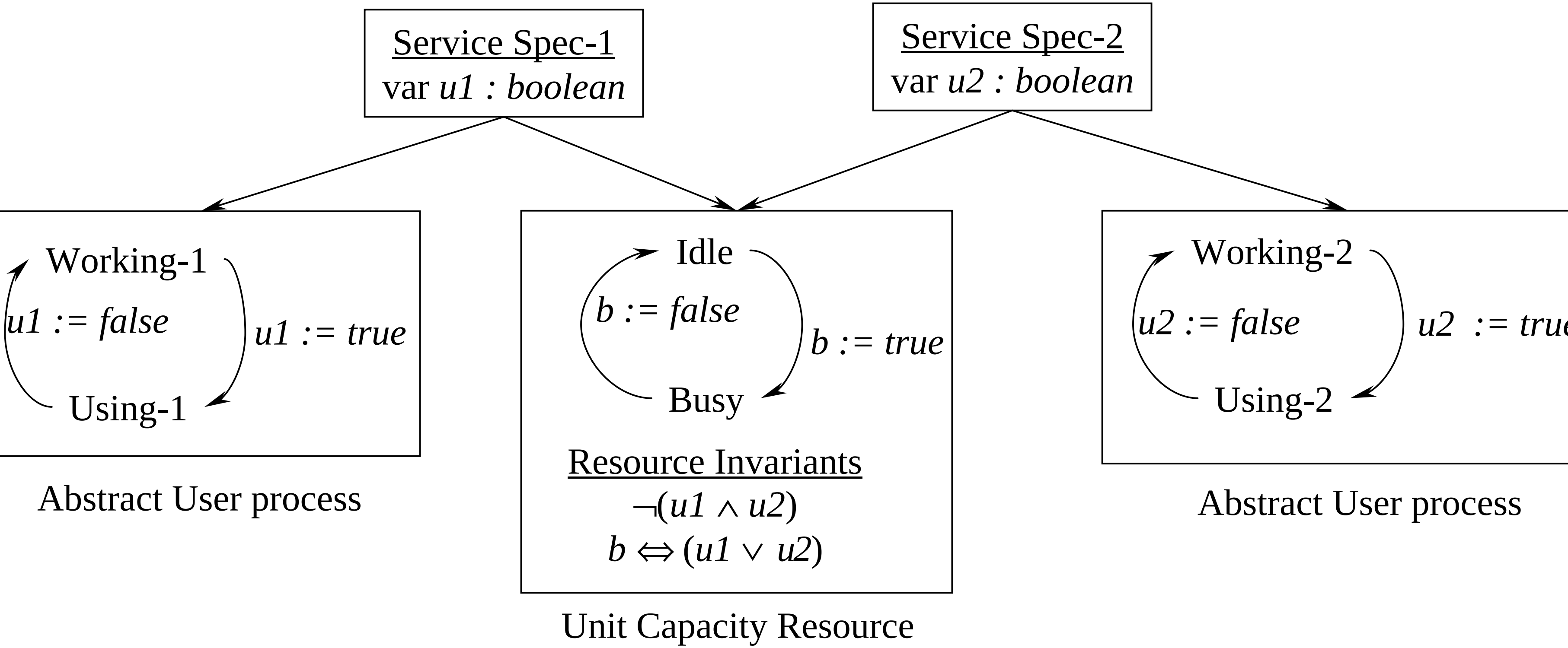


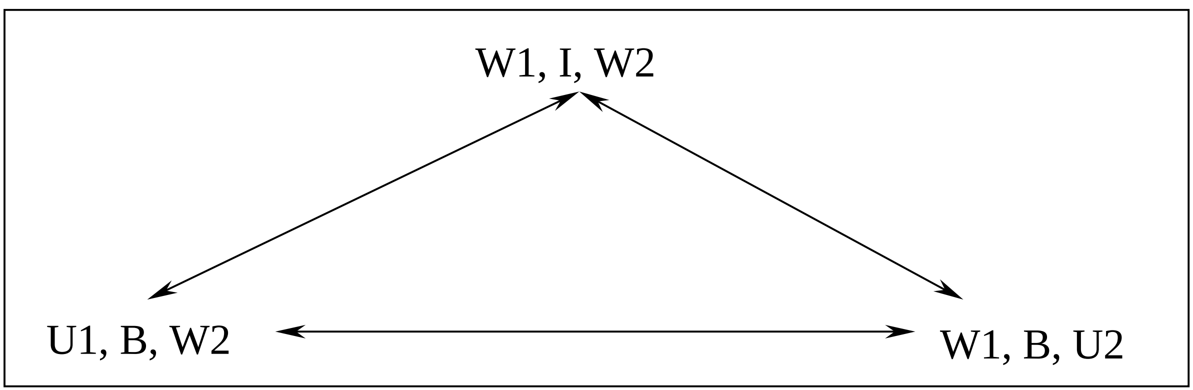
Kestrel Institute

Espec Pushout

- pullback of underlying shapes
- pushout of global specs
- pushout of corresponding state specs
- transitions obtained via universality

Composition of Behavior: Mutual Exclusion

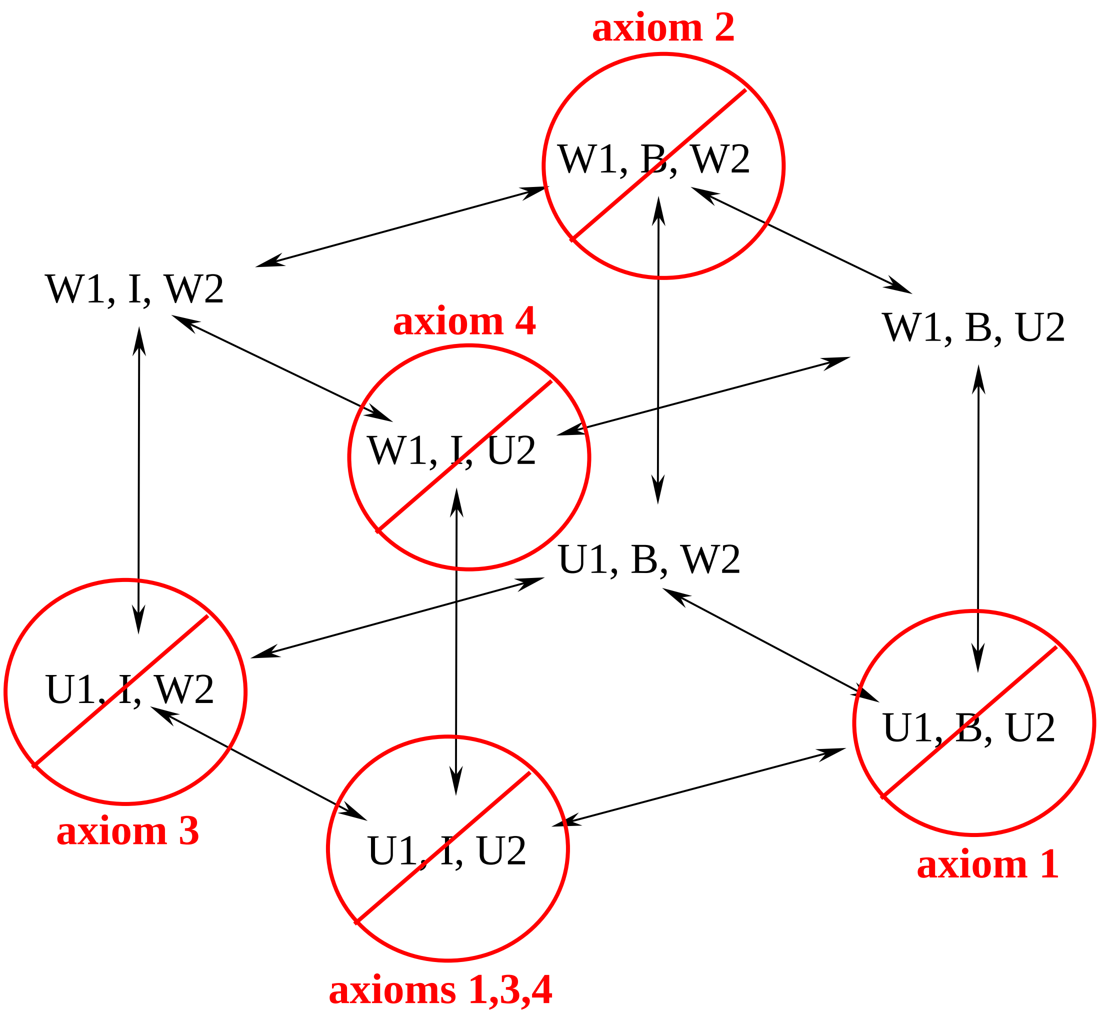




The composed espec exhibits exactly the mutual exclusive behaviors

Kestrel Institute

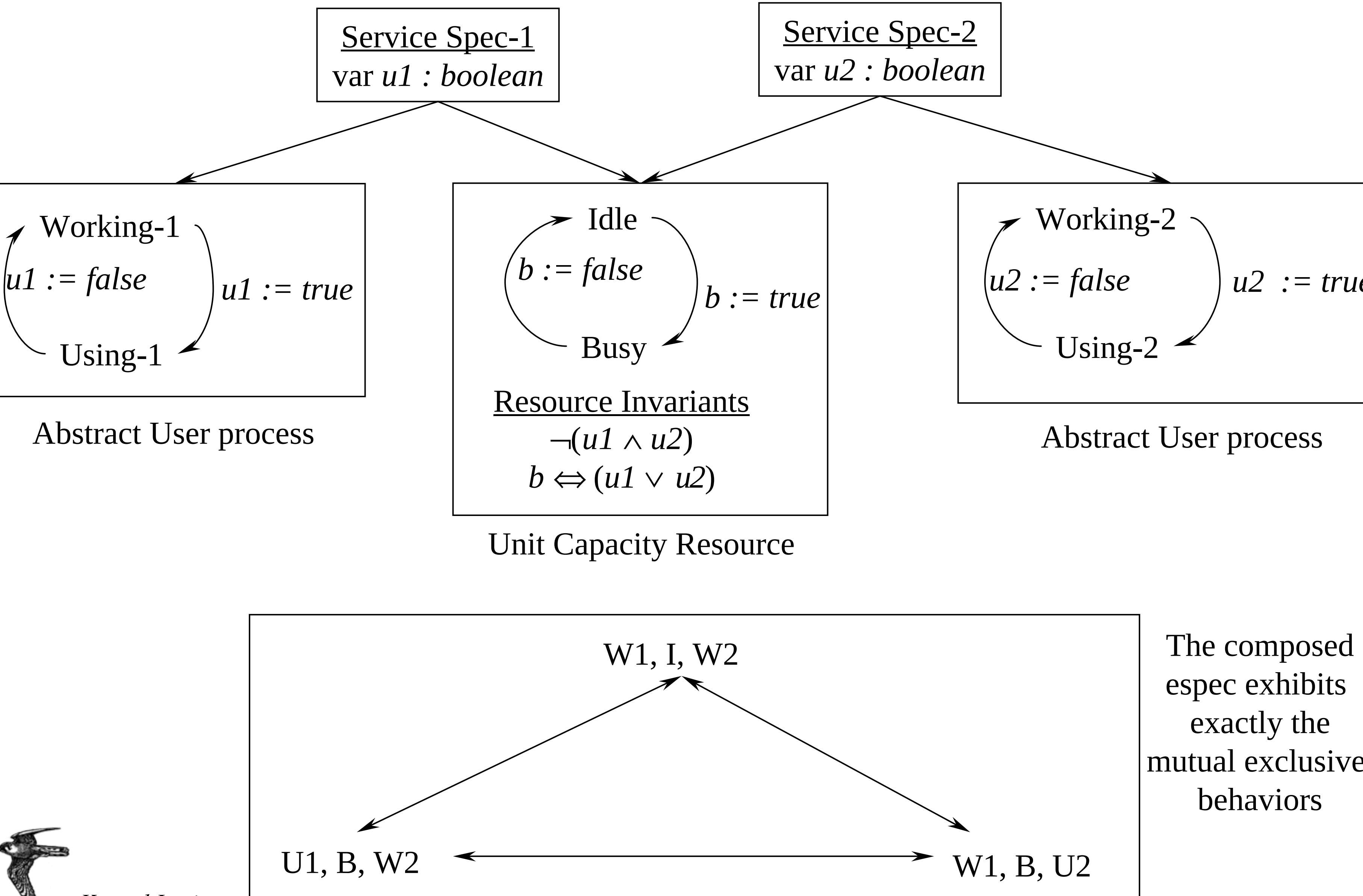
Colimit of especs



<u>Axioms</u>

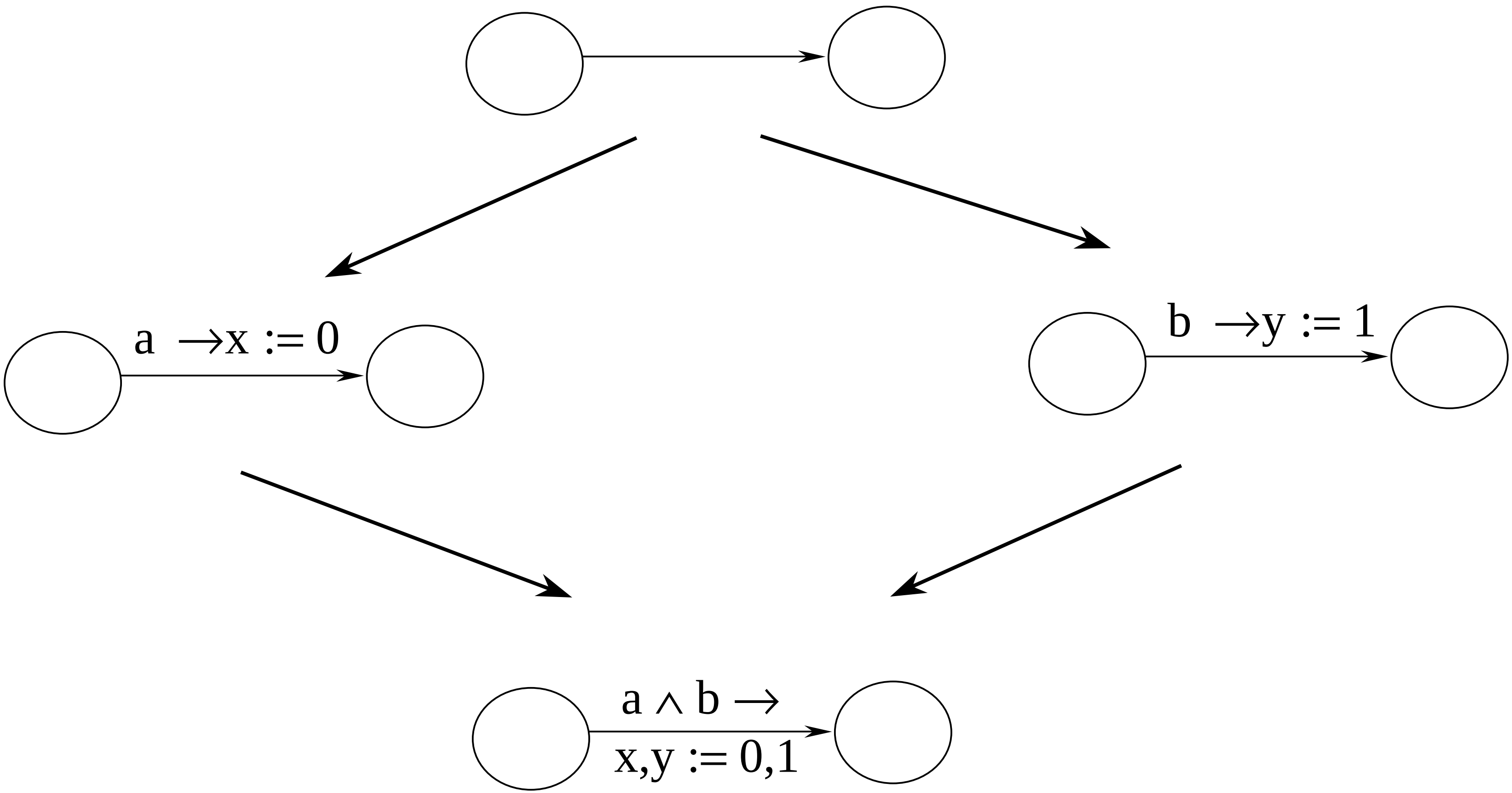
1. $\neg(u1 \wedge u2)$
2. $b \Rightarrow (u1 \vee u2)$
3. $u1 \Rightarrow b$
4. $u2 \Rightarrow b$

Composition of Behavior: Mutual Exclusion



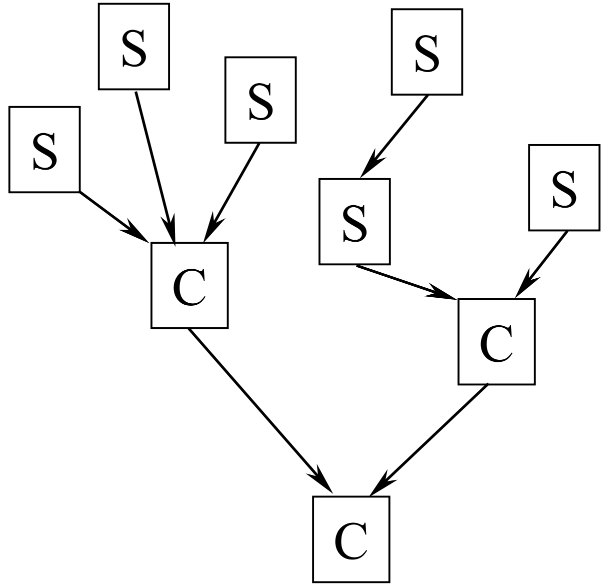
The composed espec exhibits exactly the mutual exclusive behaviors

Espec Colimit
superposition of transitions



Continuous Re-Assembly of a Sensor Network

<u>low cost "motes"</u>
4 mHz, 8-bit CPU
4 KB RAM
19.2 kbps radio links
2×2 inch
$50



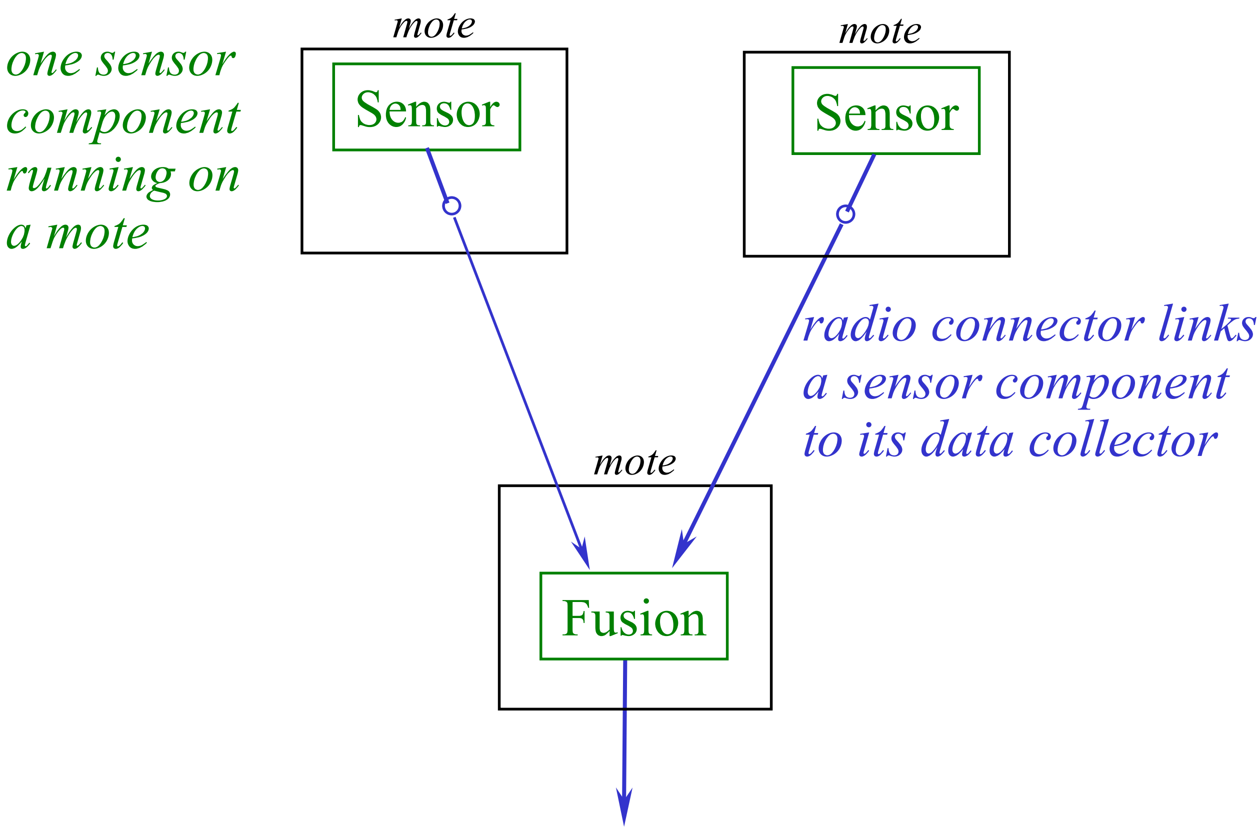
Initially, motes are randomly strewn at a site

S - sensor
C - collector

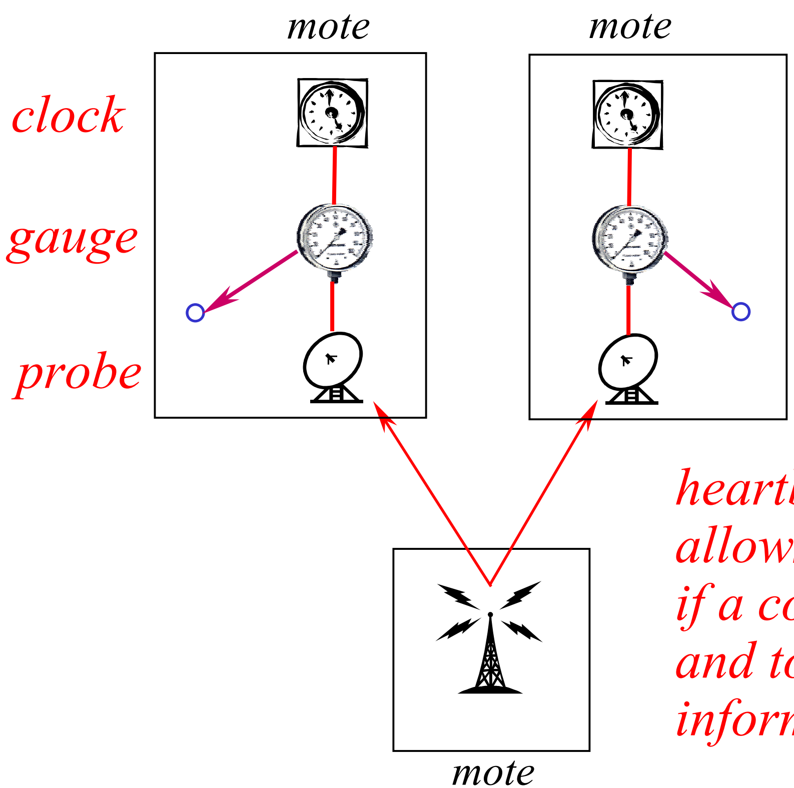
- Refinement of the logical architecture to mote components reveals unreliable communication
- At design-time, formally compose in active probes, gauges and an adaptivity scheme that at run-time re-architects the system to adapt to communication failures

Basic Architecture of a Sensor Net:

each sensor transmits data to a designated data collector

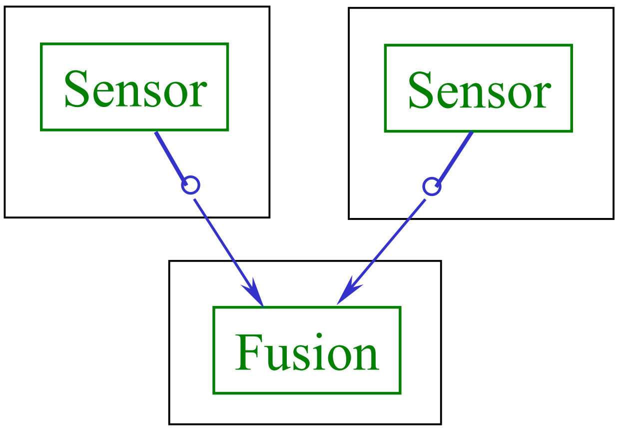


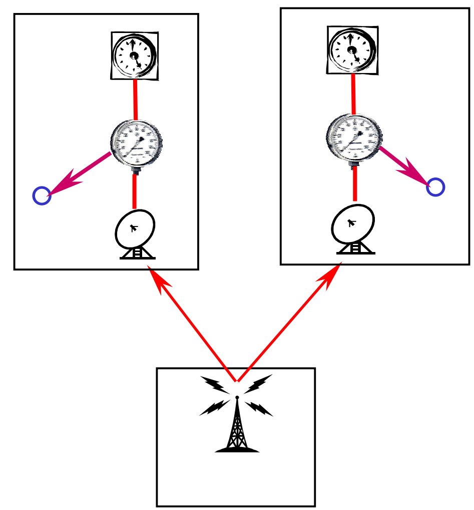
A Simple Adaptive Architecture



heartbeat probe plus clock allows gauge to decide if a connector is alive, and to send adaptation information otherwise

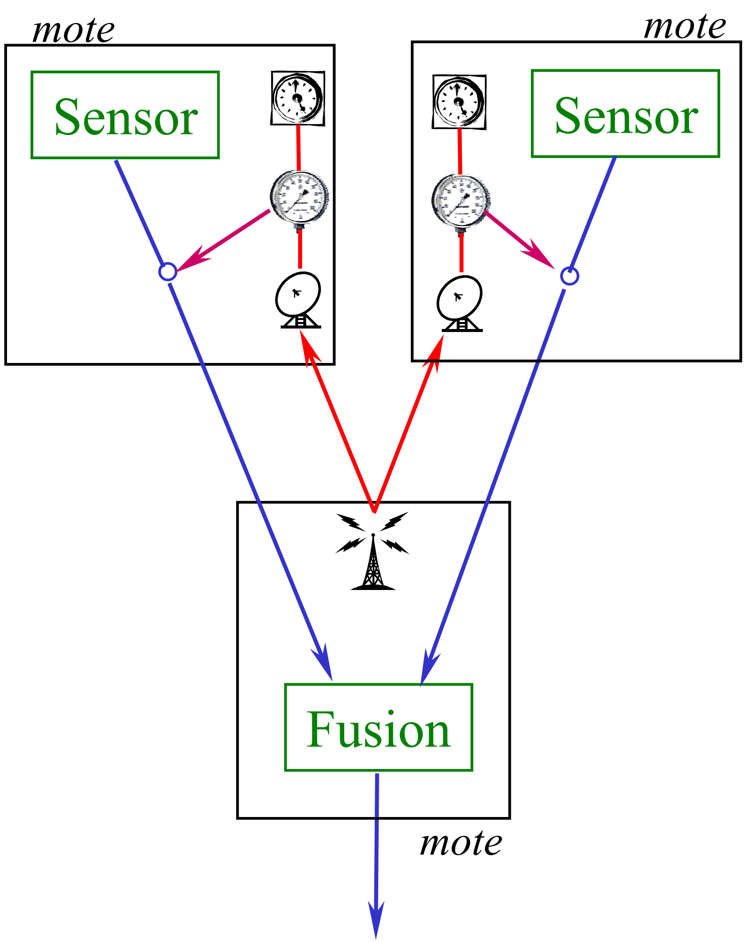
Composing Architectures:
Basic architecture is composed with adaptive architecture



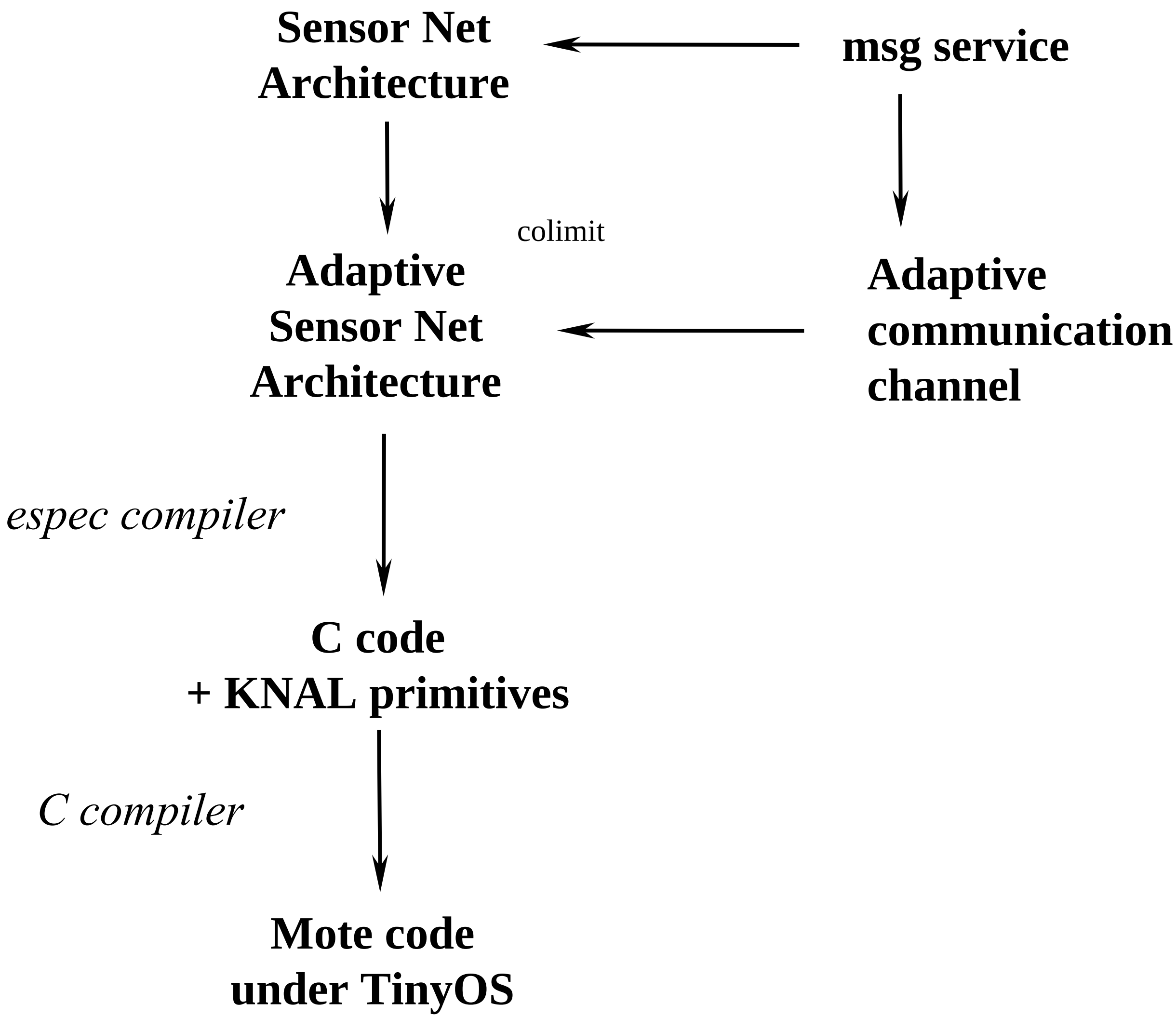


the composition is carried out
by the automatic *colimit* operation
on diagrams of especs

Result: An Adaptive Sensor Net



Refinement of a Sensor Net Architecture



Open Systems Composition
Abadi-Lamport

A system M, comprised of components $M_1, \ldots. M_n$, guarantees its services if

1. the environment satisfies its requirements E
2. M's services follow from the services provided by $M_1, \ldots. M_n$
3. each component M_i guarantees its services assuming that its environment (E + the other components) satisfies E_i

Systems Specifications
parameterization

The environment model is a *parameter* espec to an espec:

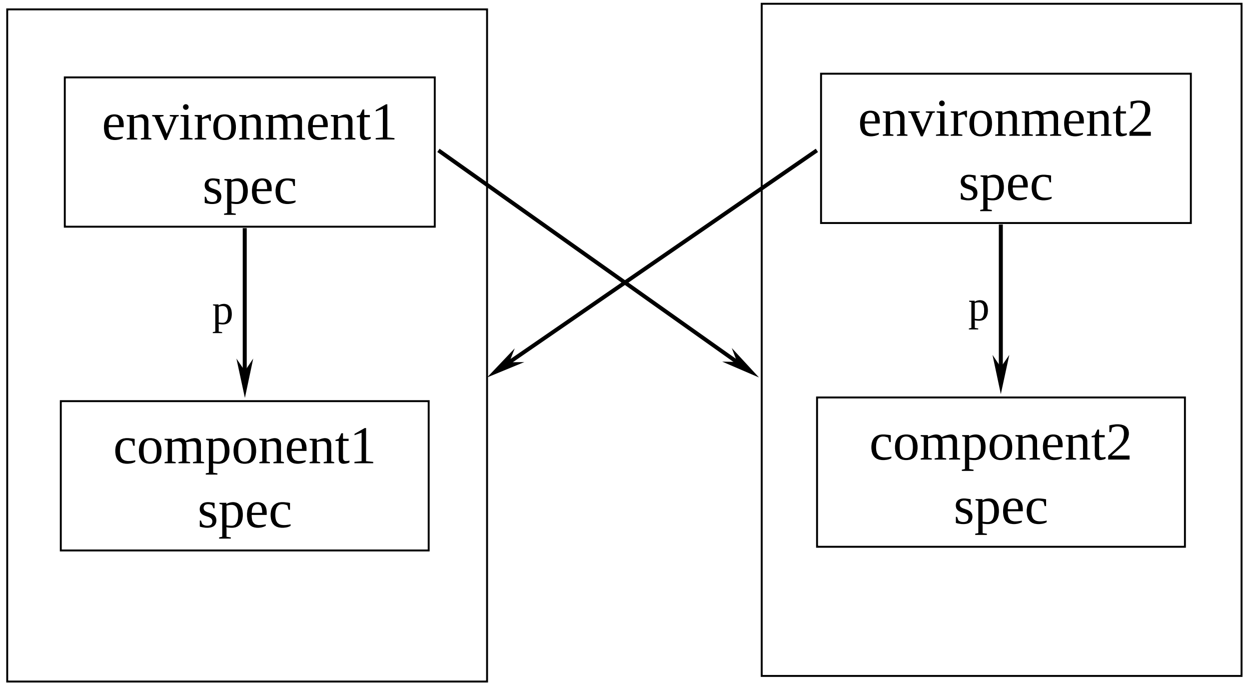
- as an espec, it can specify operations, events, invariants, timing, resource constraints

- as a parameter, the system can exploit its properties and services, but cannot refine or modify them

- the assurance arguments for the system depend on the actual environment implementing the environment model

i.e. there is a morphism from the model to the environment

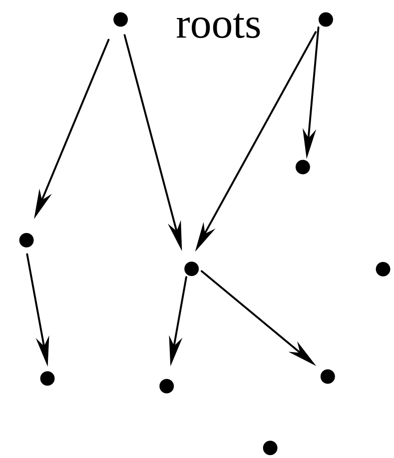
Open System Composition



Example: Concurrent Garbage Collection

Given:

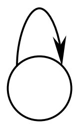
1. a finite DAG (directed acyclic graph)
2. some permanently rooted nodes
3. a Mutator process that mutates the graph
4. a Collector process that finds inaccessible nodes and makes them accessible



```
espec Directed-Rooted-Graph is
 spec
   sort Node
   sort Arc = {source : Node, target : Node}
   sort Directed-Rooted-Graph = { Nodes : set(Node),  Roots : set(Node),  Arcs  : set(Arc)
                                                    | Roots ~= {}  & Roots ⊂ Nodes  & Arcs ⊆ Nodes x Nodes }
   var G : Directed-Rooted-Graph

  function accessible? (G:Directed-Rooted-Graph, n:Node  | n in G.Nodes) : Boolean
       = (n in G.Roots  or ex(m)(<m,n> in G.Arcs & accessible?(G, m)))
 end-spec

 prog
 procedure CollectNode(n:Node | n in G.Nodes & ~accessible(G,n) ) is
   let r:Node  = some(s:Node)(s in G.Roots)
       G.Arcs := G.Arcs with (r,n) \ {<n,m> | m in G.nodes}
   postcondition: accessible(G,n)
 end-procedure

 procedure ChangeArc(a:Arc, k:Node | a in G.Arcs & k in G.Nodes & accessible(G,k)) is
   G.Arcs := (G.Arcs without a) with <a.source,k>
  end-procedure

end-espec
```

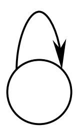
Concurrent Garbage Collection

$\neg\, acc(G,n) \Rightarrow \neg acc(G',n)$



import Directed-Rooted-Graph

$\exists\, n.\ \neg\, acc(G,n) \rightarrow \text{CollectNode}(n)$



Theorem (Safety):

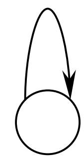
No accessible node is ever collected

Axiom (Liveness):

All inaccessible nodes are eventually collected.

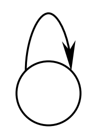
Collector

$acc(G,n) \Rightarrow acc(G',n)$



import Directed-Rooted-Graph

$\exists\, a,k.\ acc(G,k) \rightarrow \text{ChangeArc}(a,k)$



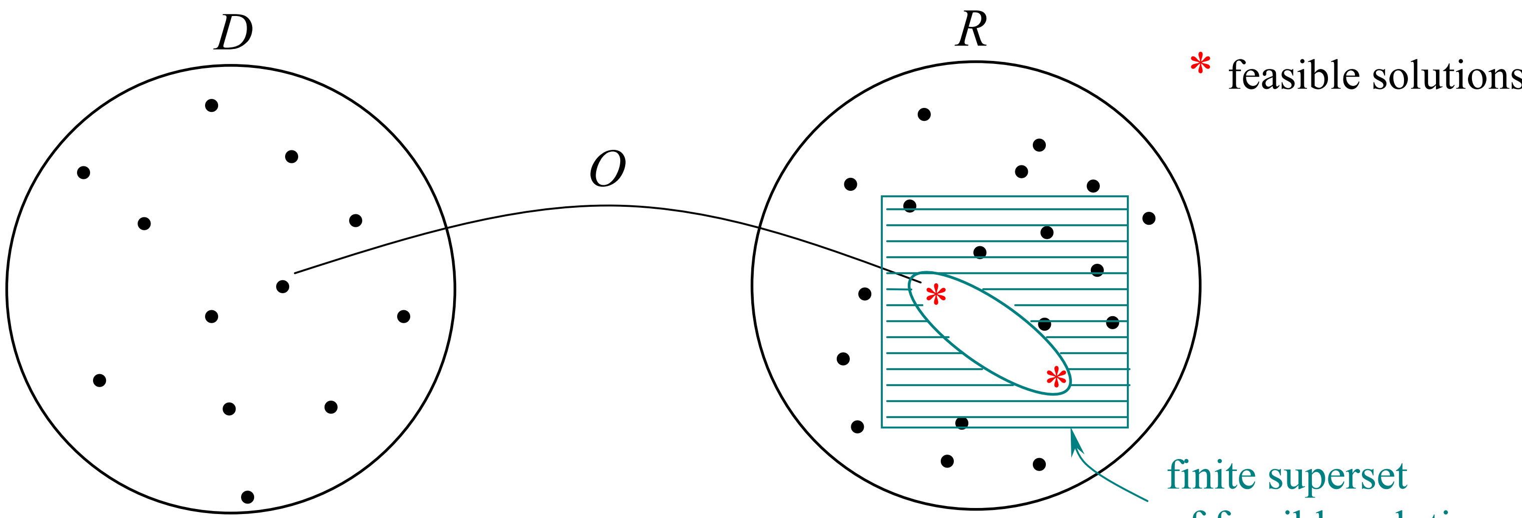
Mutator

Problem Solving Structure

Complement Reduction Structure:

finite superset of feasible solutions plus a test for nonsolutions

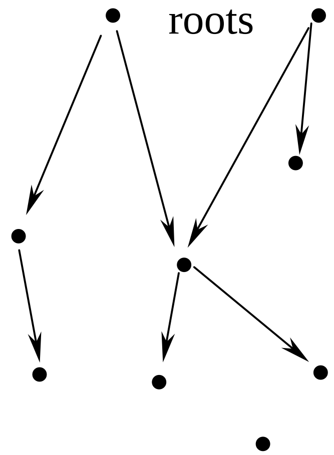
supports sieves



Sieve Example: Garbage Collection

Given:

1. a finite collection of nodes, each with some links to other nodes
2. some permanently rooted nodes



Find: all nodes inaccessible from the rooted nodes

<u>Sieve interpretation</u>

Finite superset of solutions $\mapsto$ all nodes

nonsolutions $\mapsto$ accessible nodes

<u>Sieve algorithm scheme</u>

1. mark all nodes white
2. mark all accessible nodes black
3. collect the remaining white nodes

<u>Accessibility:</u>

$n \in G.roots \Rightarrow acc(n)$

$acc(n) \wedge \langle n,m \rangle \in G.arcs \Rightarrow acc(m)$

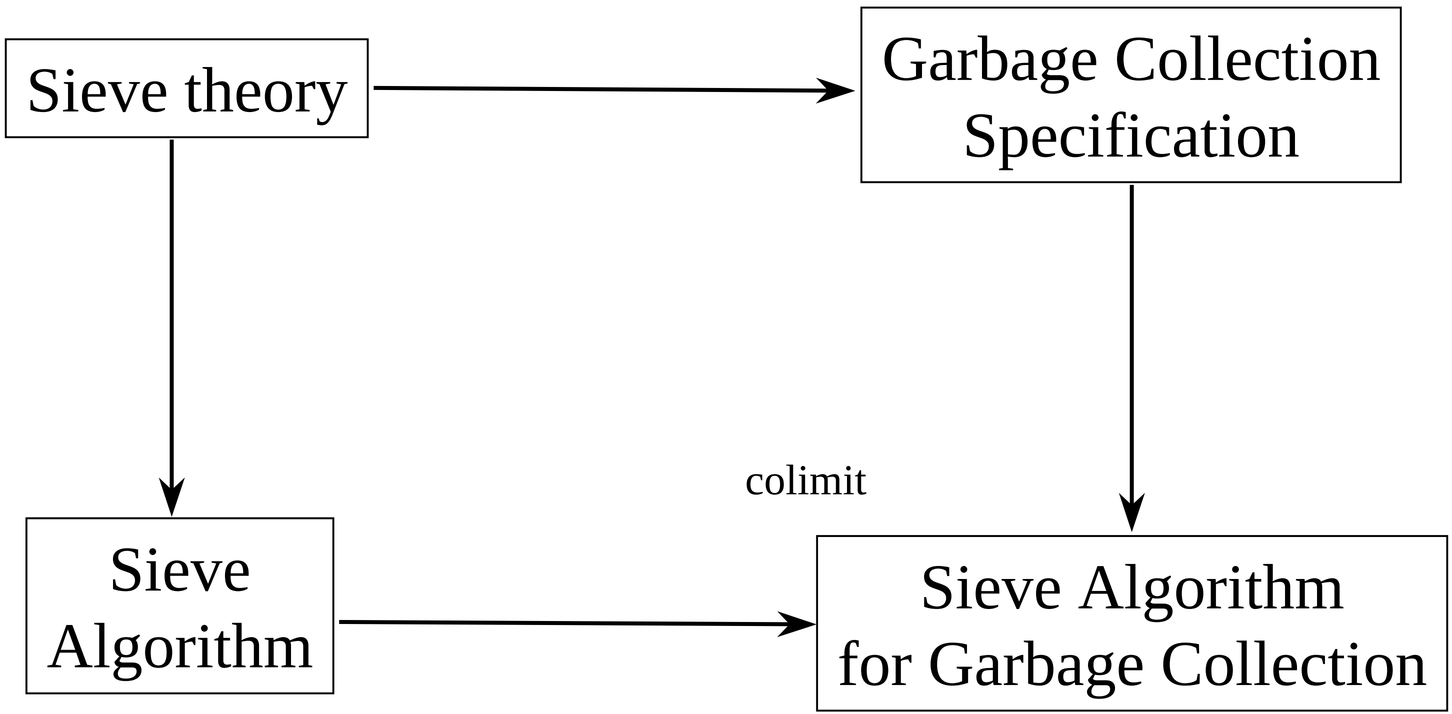
<u>Primality:</u>

$prime?(2)$

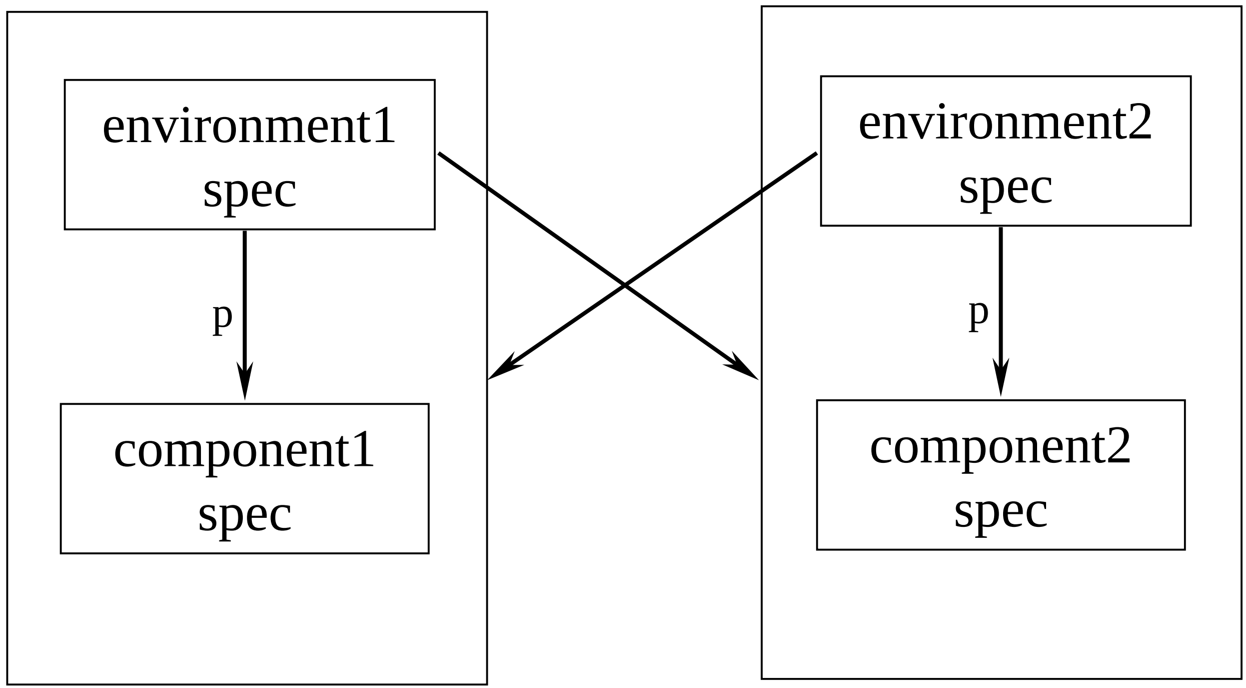
$prime?(n) \wedge plural?(i) \Rightarrow \neg prime?(n*$

Kestrel Institute

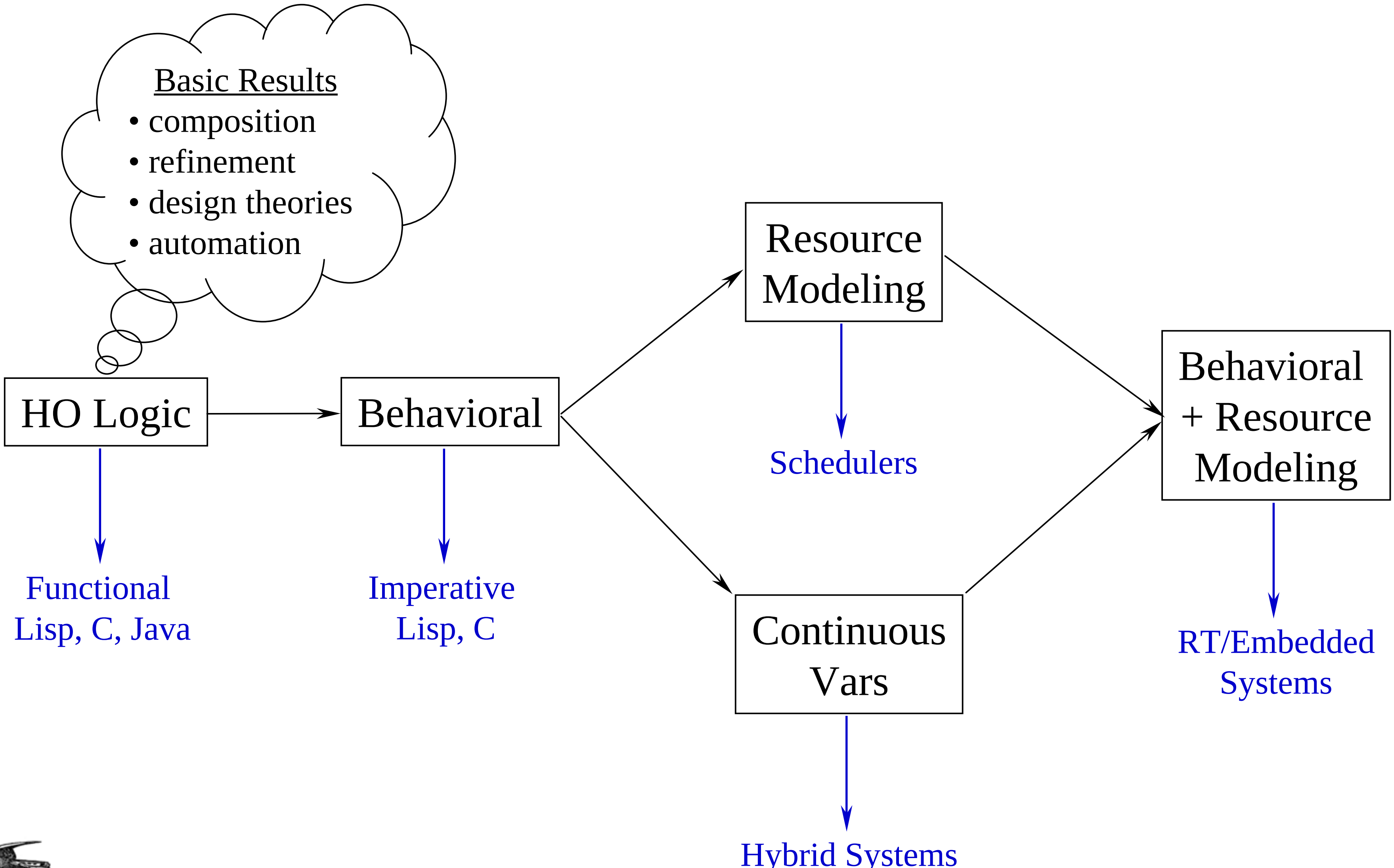
Using a Design Theory and Colimit to Construct a Refinement



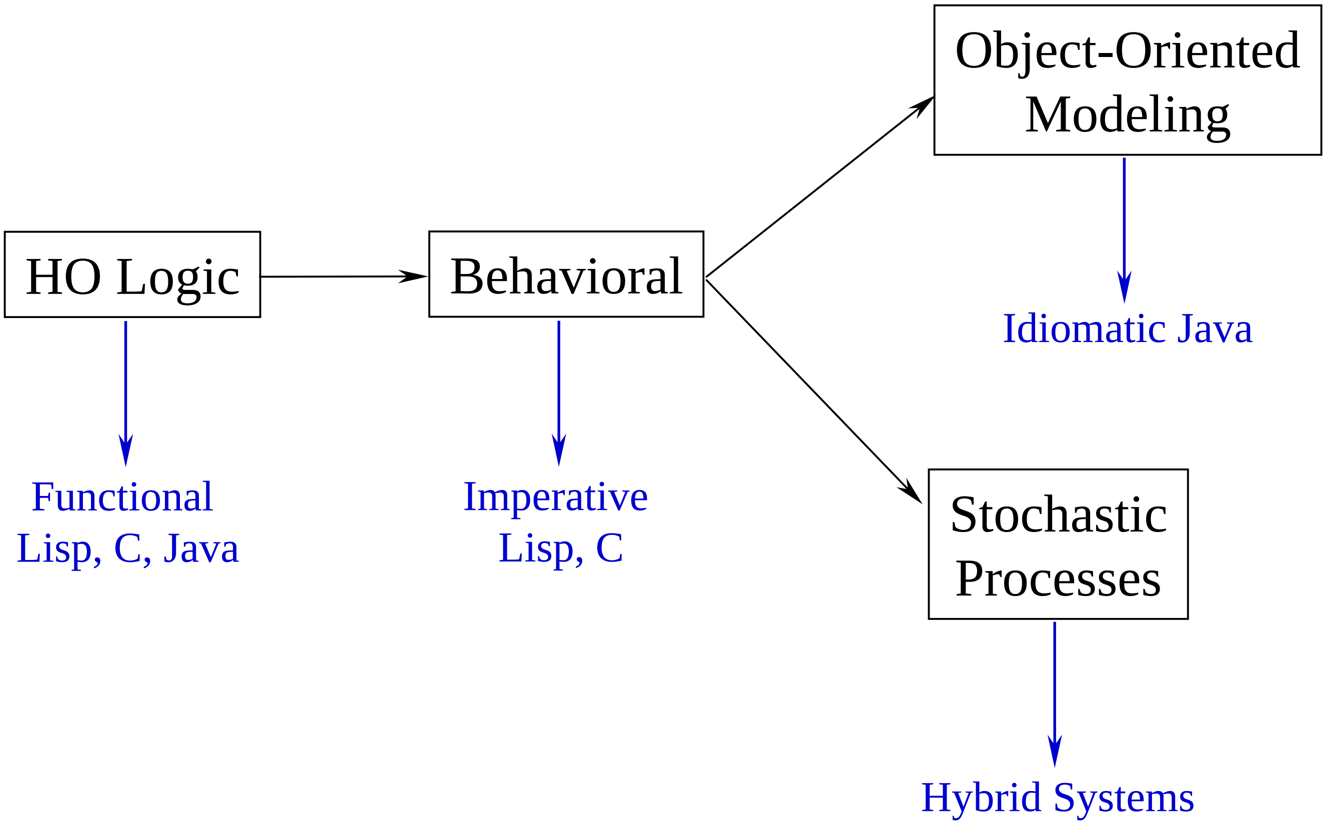
Open System Composition



A Road Map – Specification Expressiveness



A Road Map – Specification Expressiveness



Extras

Calculating a Colimit in **SPEC**

spec BINARY-RELATION is
  sort E
  op $\_br\_ : E, E \rightarrow Boolean$
end-spec

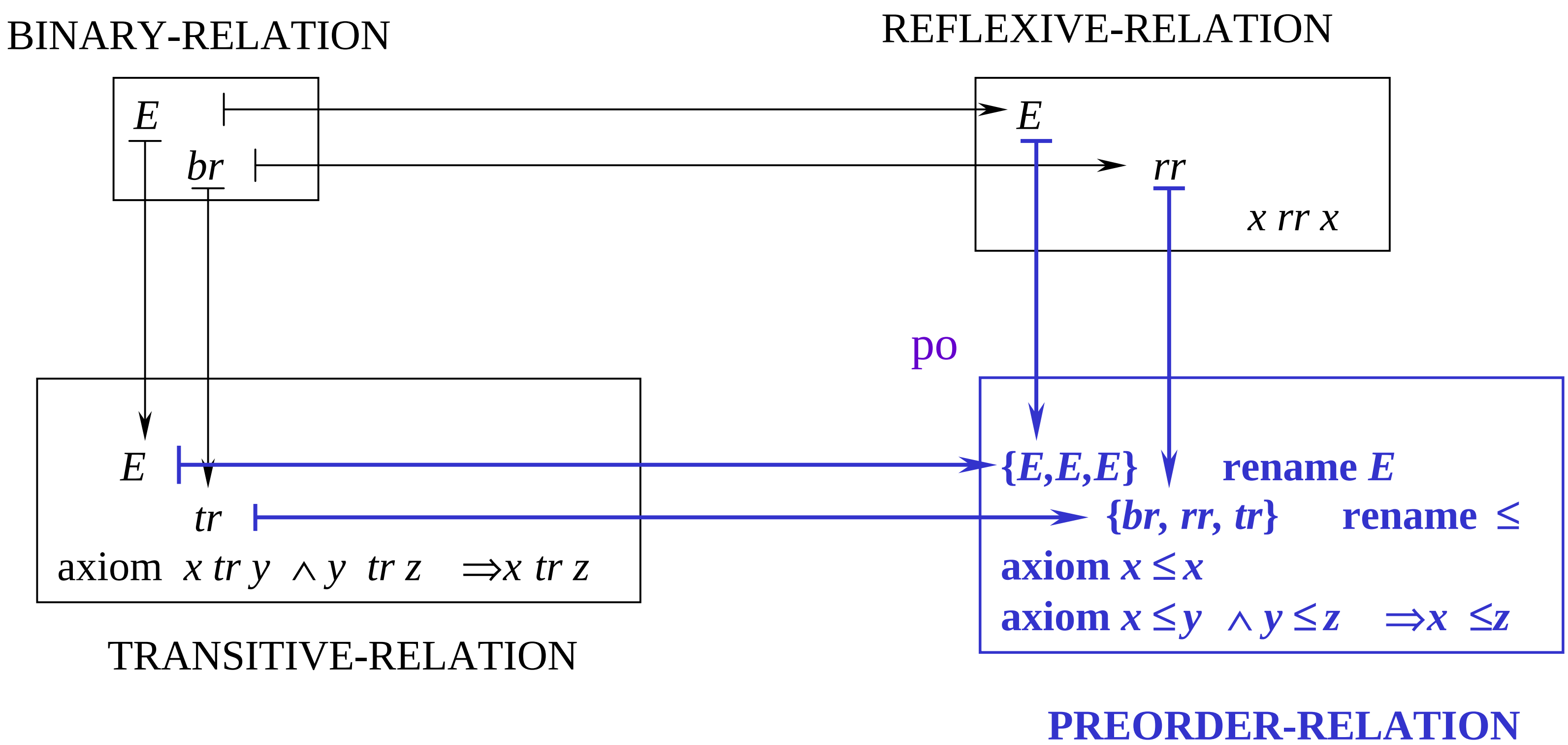
spec REFLEXIVE-RELATION is
  sort E
  op $\_rr\_ : E, E \rightarrow Boolean$
  axiom reflexivity is $a\ rr\ a$
end-spec

spec TRANSITIVE -RELATION is
  sort E
  op $\_tr\_ : E, E \rightarrow \text{Boolean}$
  axiom transitivity is
    $a\ tr\ b \wedge b\ tr\ c \Rightarrow a\ tr\ c$
end-spec

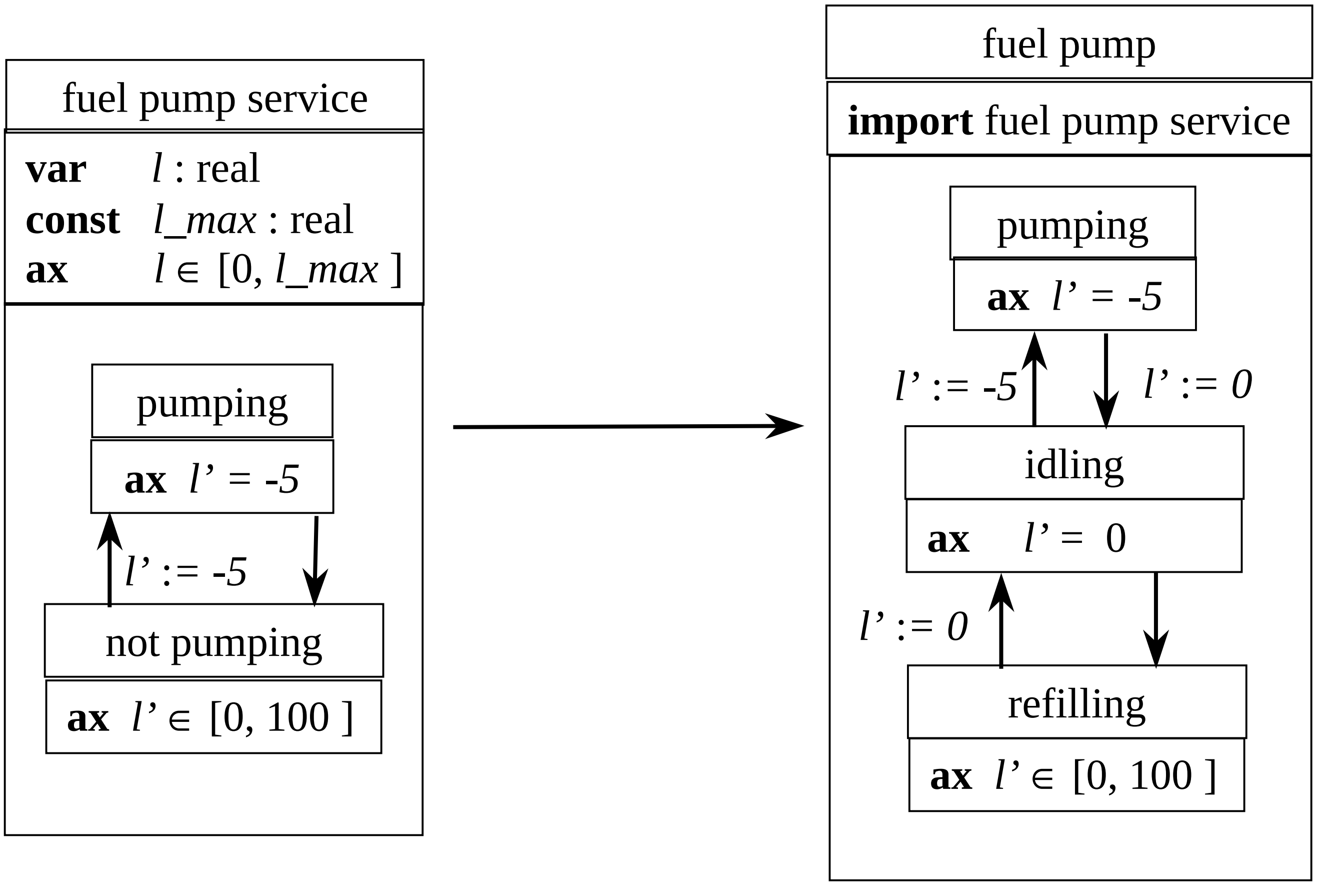
spec PREORDER-RELATION is
  sort E
  op $\leq : E, E \rightarrow Boolean$
  axiom reflexivity is
    $a \leq a$
  axiom transitivity is
    $a \leq b \wedge b \leq c \Rightarrow a \leq c$
end-spec

Calculating a Colimit in **SPEC**

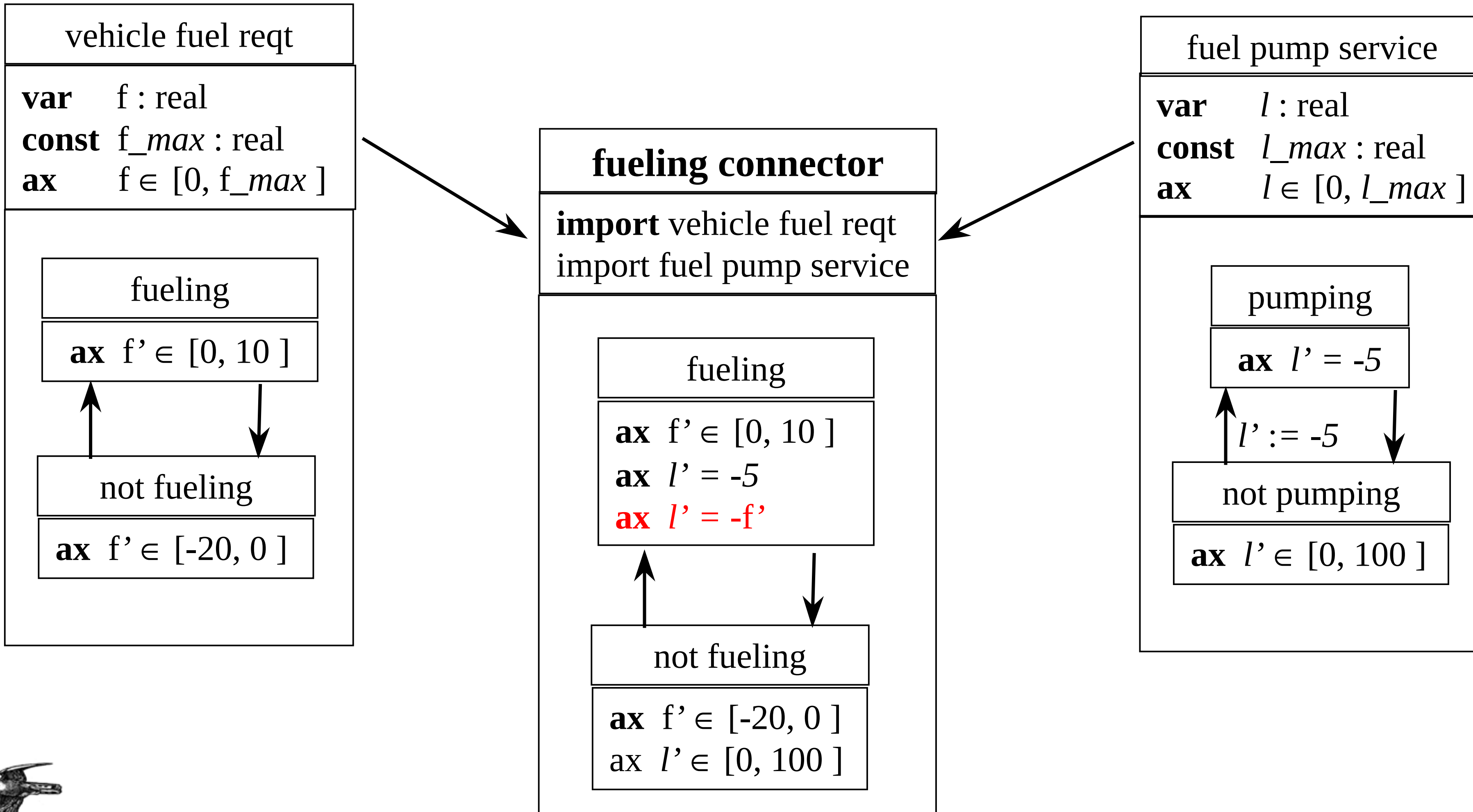
Collect equivalence classes of sorts and ops from all specs in the diagram.



Hybrid Especs
modeling continuous behavior



Hybrid Especs
modeling continuous behavior



Kestrel Institute