



COSBI

Programming-based systems biology

BlenX Language



the development of the appropriate languages to describe information processing in biological systems and the generation of more effective methods to translate biochemical descriptions into the functioning of the logic circuits that underpin biological phenomena

**Paul Nurse**

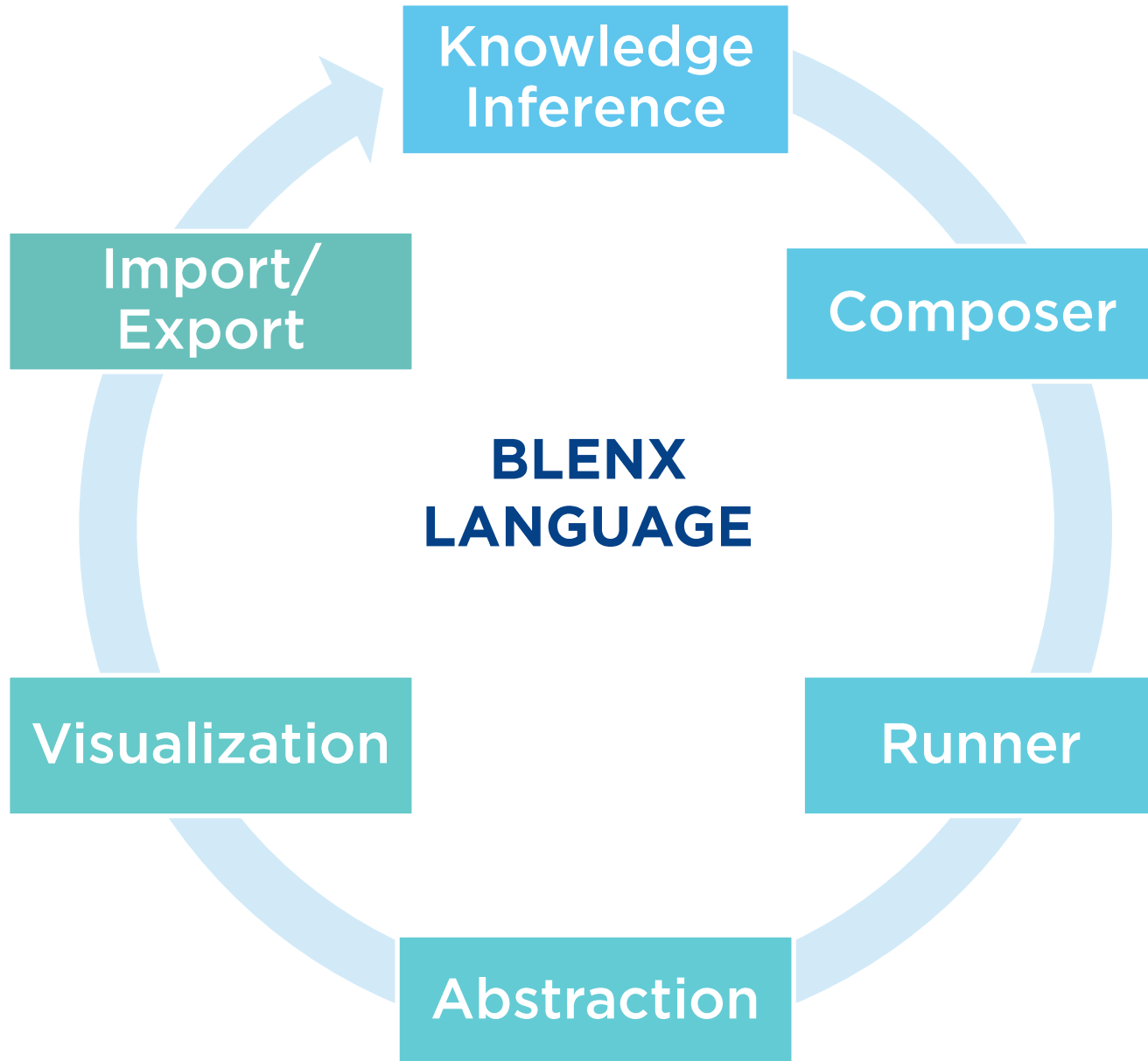
# THE METAPHOR

---

biological entities	processes
interaction capabilities	interfaces
complex/decomplex	binding/unbinding
dynamics	state change

# THE ENVIRONMENT

---



# BLENX GENESIS

Case  
Studies

L. Dematté, C. Priami, A. Romanel. *The BlenX Language: a tutorial*. In Formal Methods for Computational Systems Biology, (M. Bernardo, P. Degano, G. Zavattaro, Eds.), LNCS 5016, 313-365, Springer, 2008.

L. Dematté, C. Priami, A. Romanel. *The Beta Workbench: a computational tool to study the dynamics of biological systems*. *Briefings in Bioinformatics*, 9(5): 437-449, 2008.

L. Dematté, R. Larcher, A. Palmisano, C. Priami, A. Romanel. *Programming Biology in BlenX*. In Systems Biology for Signaling Networks 1:777-821, S. Choi, Ed., Springer, 2010.

C. Priami. *The Stochastic pi-calculus*. *The Computer Journal*, 38(6):578-589, 1995.

C. Priami, A. Regev, E. Shapiro, W. Silvermann. *Application of a stochastic name-passing calculus to representation and simulation of molecular processes*. *Information Processing Letters*, 80:25-31, 2001.

C. Priami. *Language-based performance prediction of distributed and mobile systems*. *Information and Computation*, 175:119-145, 2002.

Stochastic  
pi-calculus  
1995

Beta  
binders  
2004

C. Priami and P. Quaglia. *Beta binders for biological interactions*. Proceedings of CMSB04, LNBI 3082, 21-34, 2005.

C. Priami, P. Quaglia. *Operational patterns in Beta-binders*. *Transactions on Computational Systems Biology*, 1:LNBI 3380, 50-65, 2005.

BlenX  
2008

L  
2011

Implementation

# PI-CALCULUS

---

## Processes

- Domains, Molecules/Proteins, Systems

## Primitives

- Join/Split, Conditional Events
- Restriction for Compartments, Membranes, Complexes

## Complex/Decomplex

- Not explicitly modeled
- Interplay between private names and scope size

## Low-level programming

- Reversibility of interactions

## Modeling

- Emergent behavior must be programmed

## Interaction

- Key-lock mechanism

## Implementation

- General structural congruence

# BETA-BINDERS

---

## Processes

- Domains, Molecules/Proteins, Systems

## Primitives

- Join/Split, Conditional Events
- Restriction for Compartments, Membranes, Complexes

## Complex/Decomplex

- Not explicitly modeled
- Interplay between private names and scope size

## Low-level programming

- Reversibility of interactions

## Modeling

- Emergent behavior must be programmed

## Interaction

- Key-lock mechanism

## Implementation

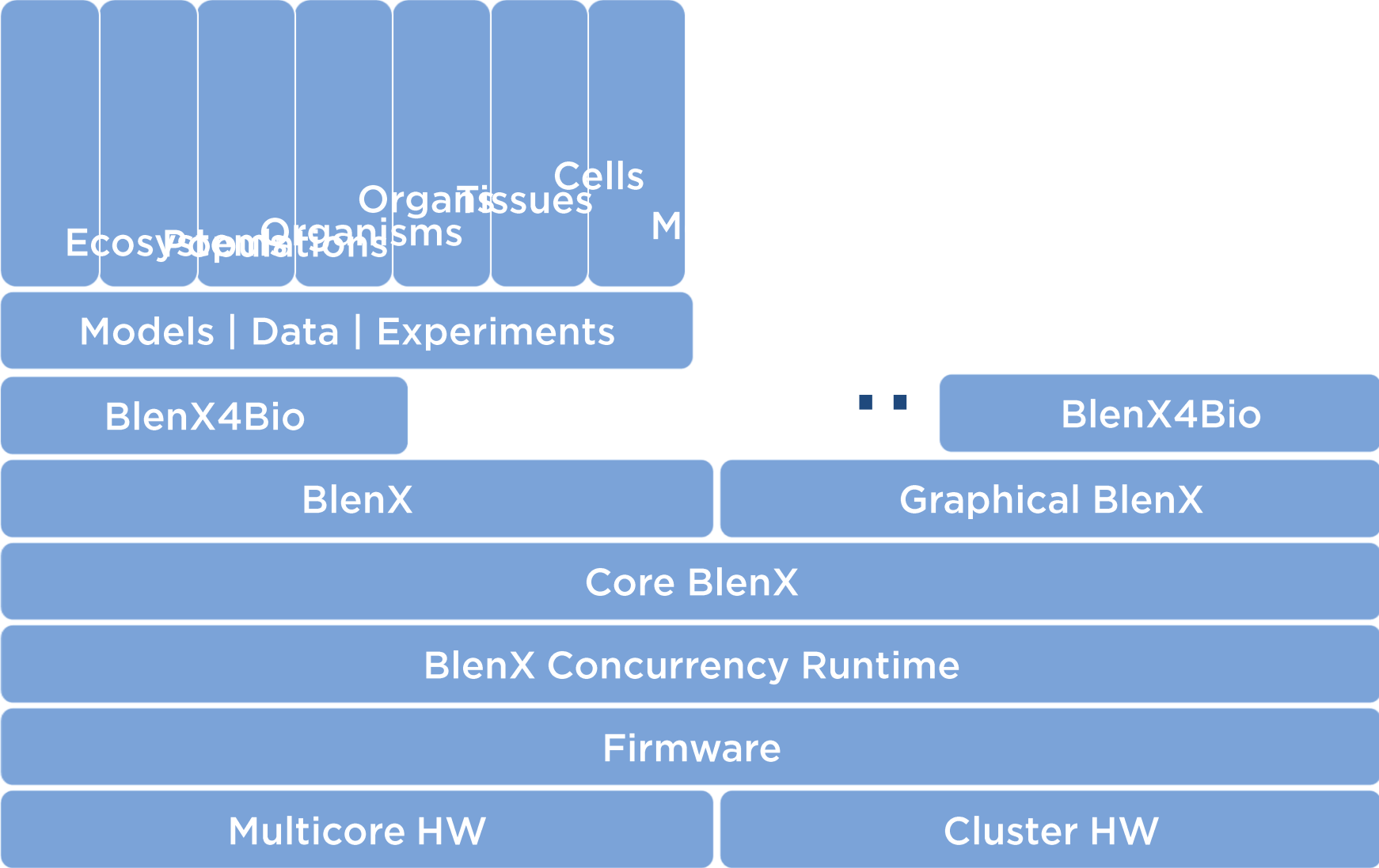
- General structural congruence

# BLENX PRINCIPLES

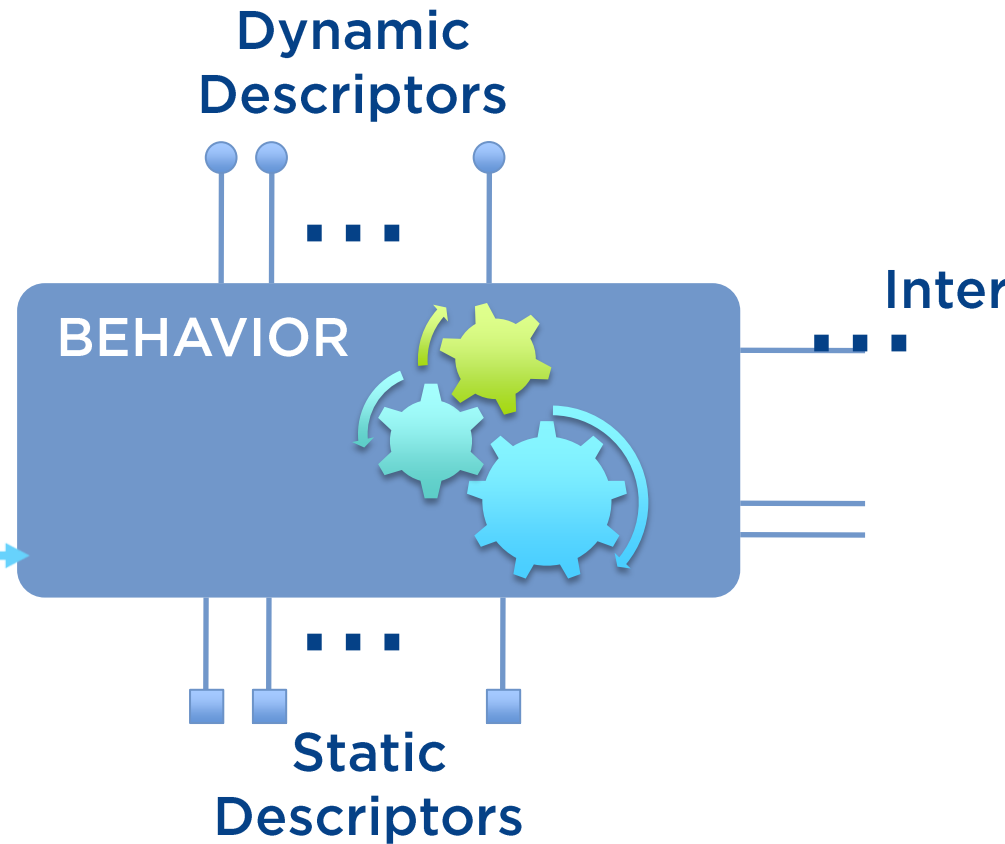
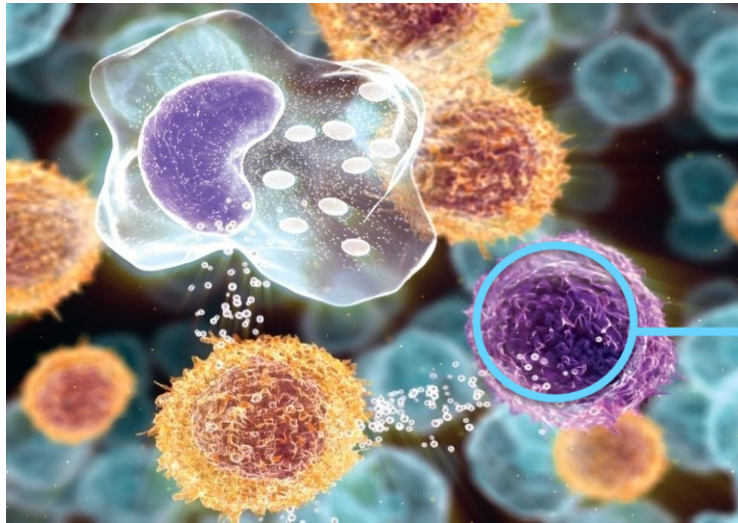
---

- Set of interacting boxes with internal behavior
- Non key-lock interaction mechanism
- Complex description and their Dynamic generation
- Typed and dynamic varying interfaces
- 1-1 correspondence with Biological elements
- Events

# BLENX TOWER



# BLENX CORE ELEMENTS



C. Priami, P. Quaglia and A. Romanel. *BlenX - static and dynamic semantics*. Proceedings of CONCUR09, LNCS 5710, 37-52, Springer, 2009.

A. Romanel. Dynamic Biological Modelling: a language-based approach.  
PhD Thesis. COSBI TR PhD-1-2010.

# BLENX PROGRAMMING LANGUAGE

---

```
// CDP
let CDP_ : bproc = #(y,CDP) [ rep start?().Prog_CDP | Prog_CDP ];
let dCDP_ : bproc = #(y,dCDP) [ rep start?().Prog_CDP | Prog_CDP ];
let dCTP_ : bproc = #(y,dCTP) [ rep start?().Prog_CDP | Prog_CDP ];
// RR
let RR_ : bproc = #(r,RR) [ rep start?().Prog_RR | Prog_RR ];
// CELL
// f1
let Cell : bproc = #(cell,Alive) [ nil ];
when ( Cell :: cell_death_function ) delete(1);
run 100 CDP_ || 100 dFdC_ || 1000 CDA_ || 1000 RR_ || 100 Cell || 100 DNA_healthy
```

# LANGUAGE FLAVOR: BLENX

```
// Tree.prog

[ steps = 10, delta = 10 ]

<< BASERATE:inf, HIDE:inf, UNHIDE:inf, CHANGE:inf >>

// Initiator Definition
let Initiator : bproc = #(out,l) [ out?().out!(root).nil ];

// Node Definition
let p1 : pproc = !out1?().out2?().out1!(node).out2!(node).nil ;
let p2 : pproc = !out1?().out2?().inp!().inp?(m).out1!(m).out!(m).nil ;
let nodeP : pproc =
  inp!().inp?(t).( t!() | (
    node?().unhide(out1).unhide(out2).p2 +
    root?().unhide(out1).unhide(out2).p1
  ) );

let Node : bproc = #(inp,IN),#h(out1,ONE),#h(out2,TWO)
  [ nodeP ];

// Init
run 2 Initiator || 10 Node
```

```
// Tree.types

{ I, IN, ONE, TWO }
%%
{
  (I,IN,100,0,inf),
  (ONE,IN,1,0,inf),
  (TWO,IN,1,0,inf)
}
```

a static list of the entities of  
the initial configuration

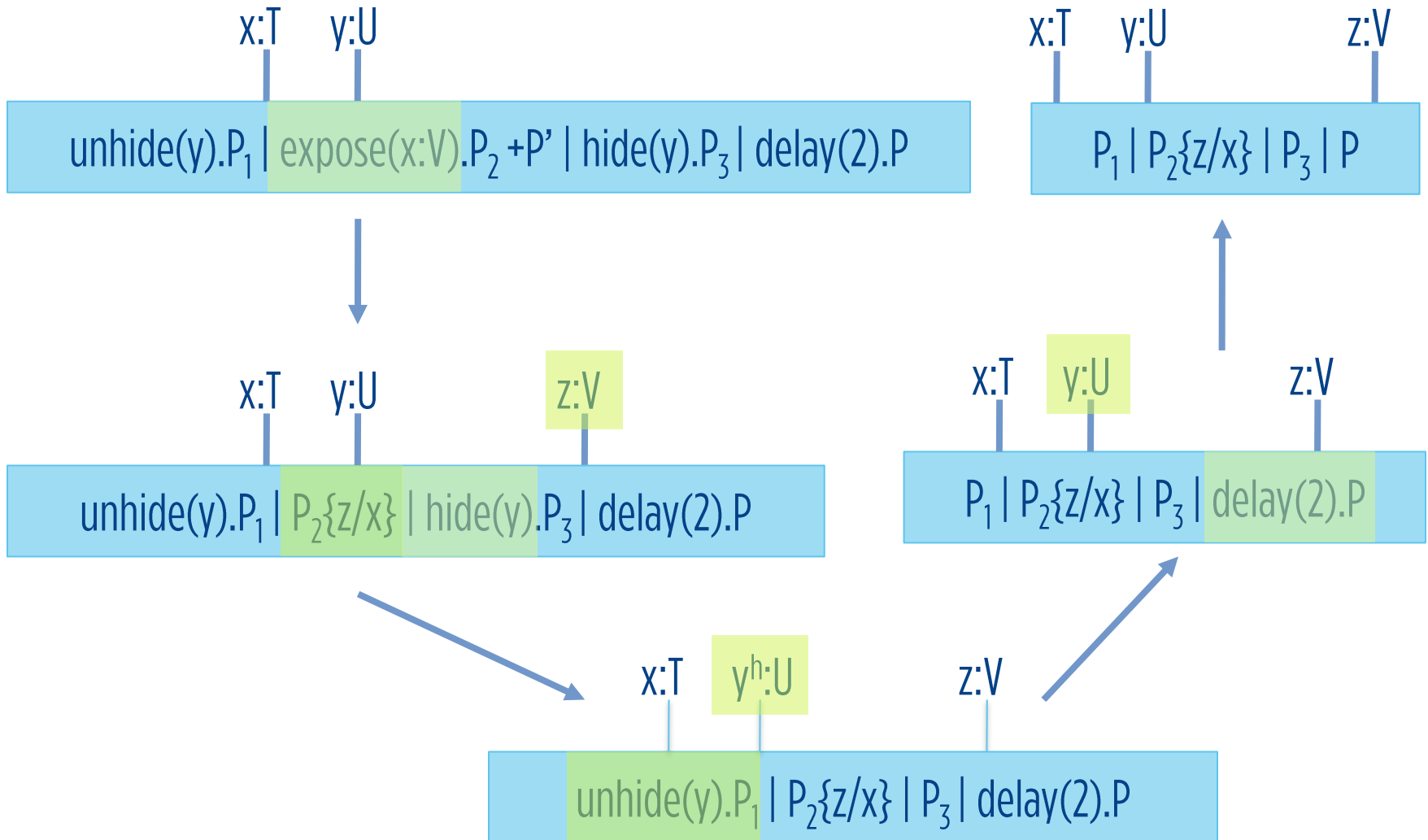
+

their amount and specificity  
for interaction.

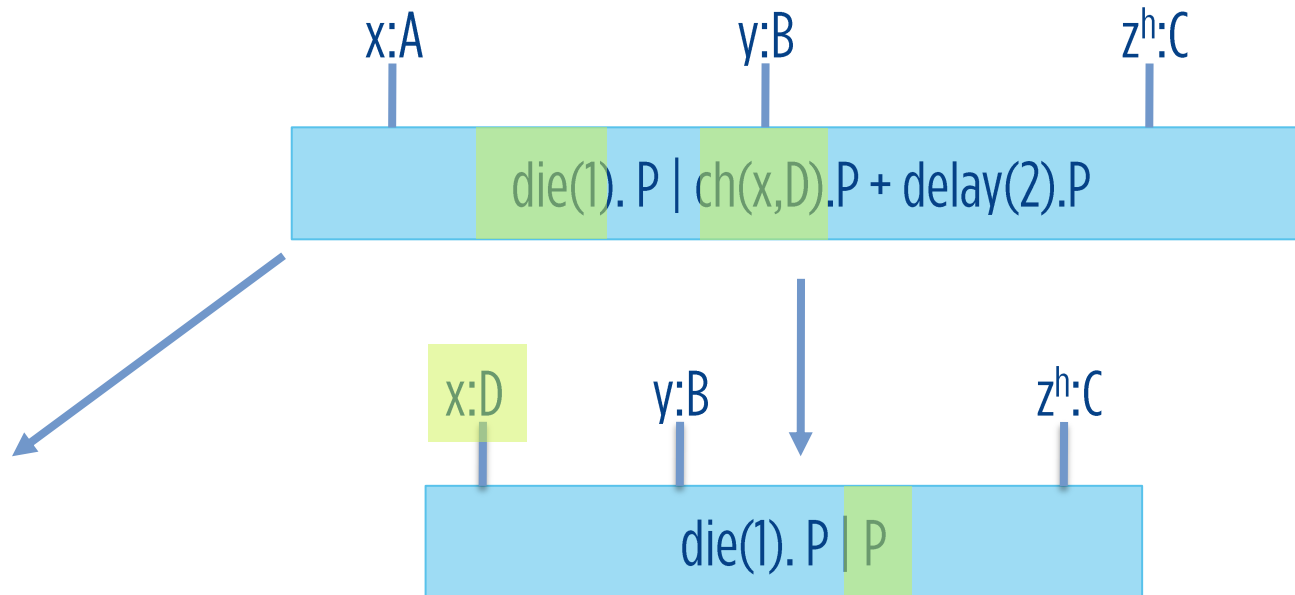
## ENABLING LIBRARY-BASED MODELING

Further examples and downloads: <http://www.cosbi.eu/index.php/research/prototypes/beta-wb>

# BLENX: MONOMOLECULAR ACTIONS

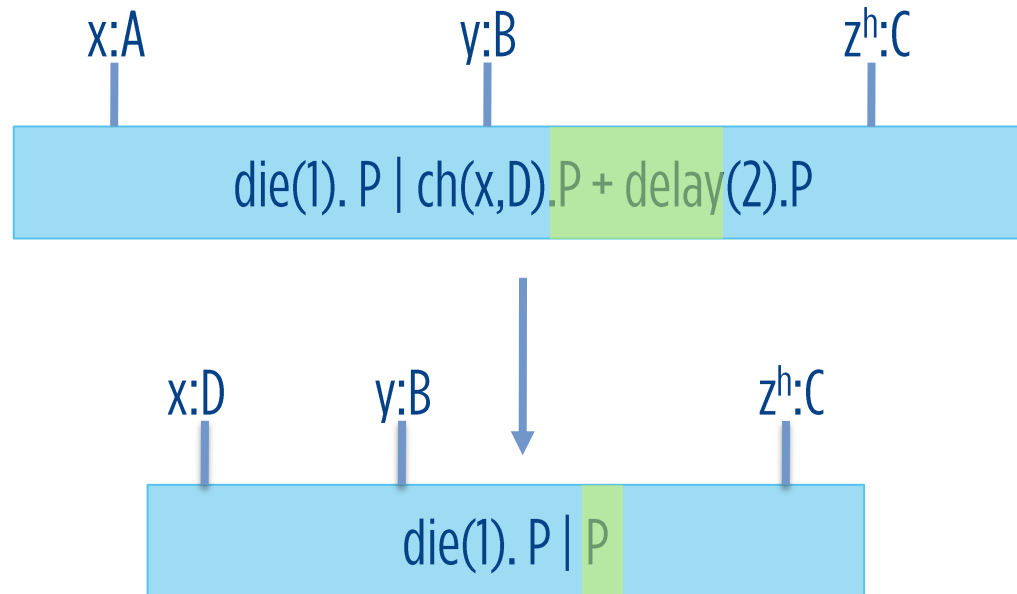


# BLENX: MONOMOLECULAR ACTIONS

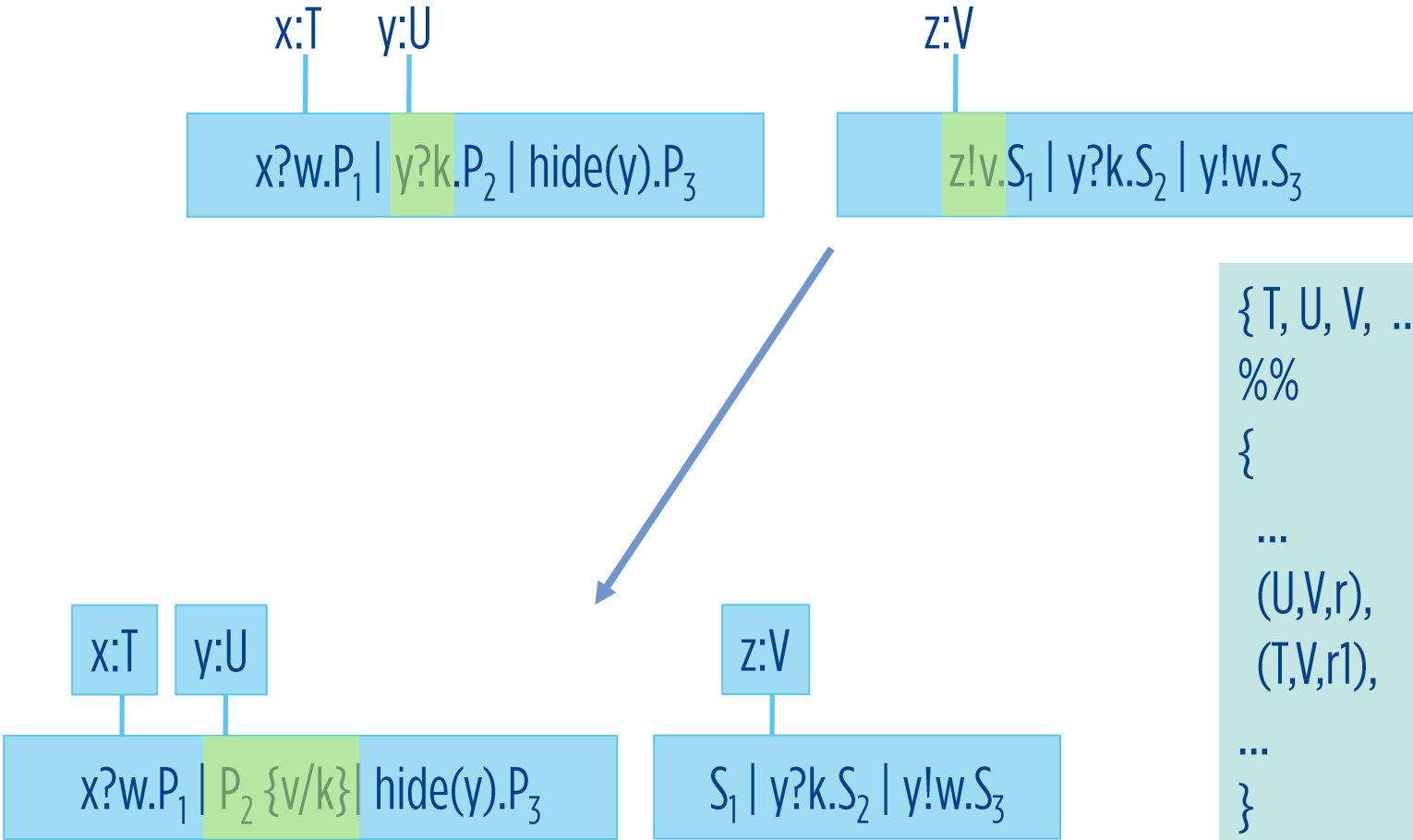


# BLENX: MONOMOLECULAR ACTIONS

---



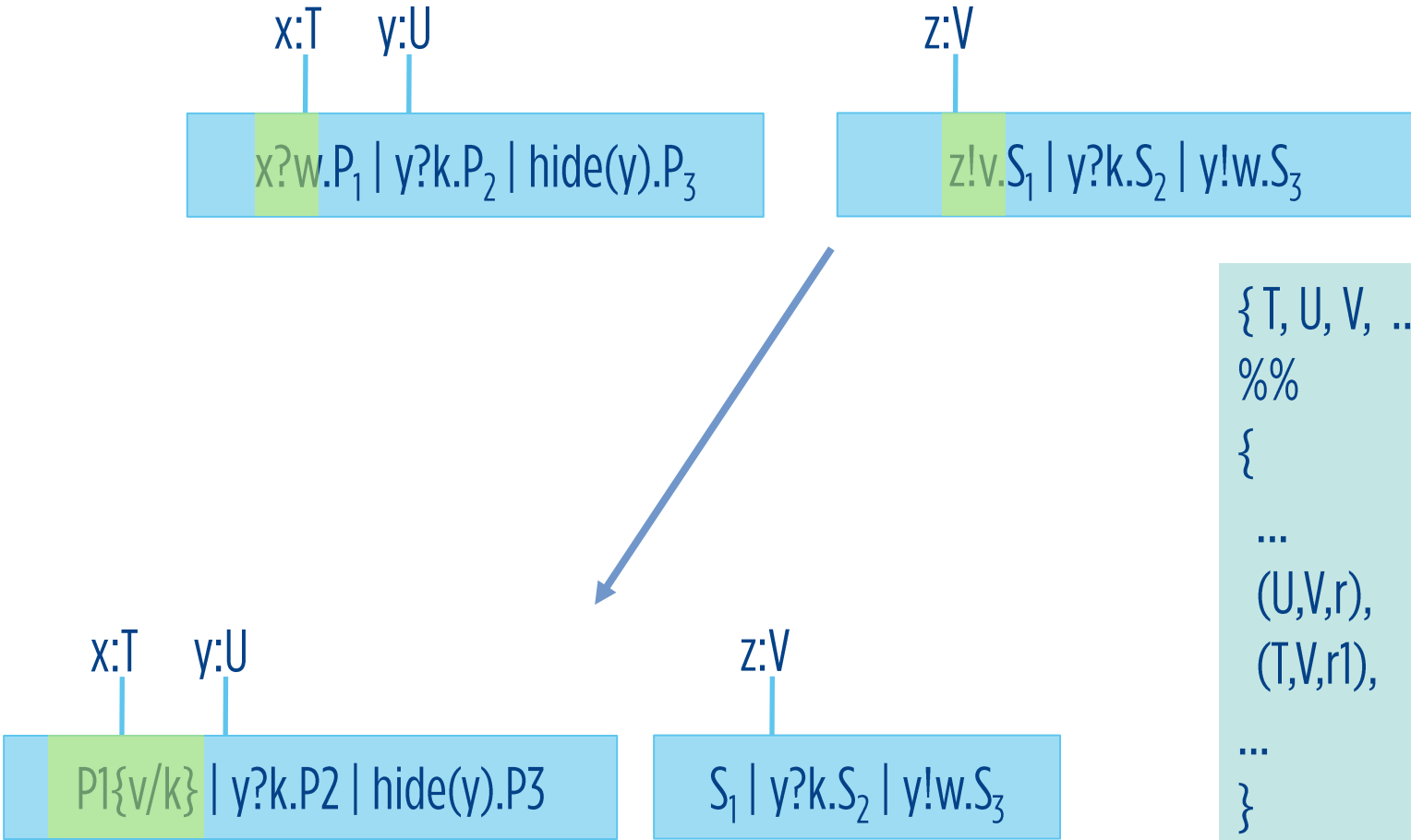
# BLENX: BIMOLECULAR ACTIONS



```

{ T, U, V, ... }
%%
{
...
(U,V,r),
(T,V,r1),
...
}
    
```

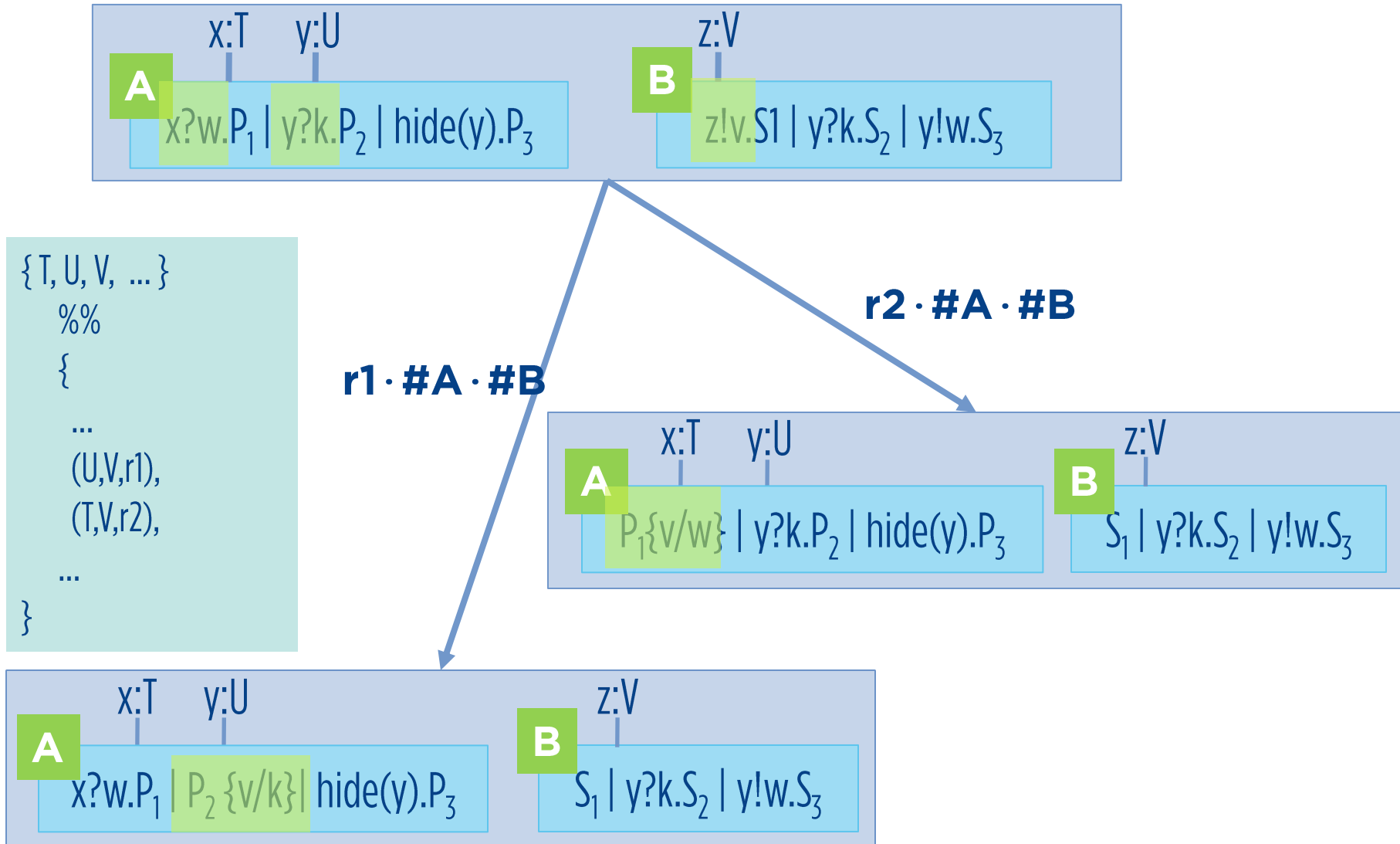
# BLENX: BIMOLECULAR ACTIONS



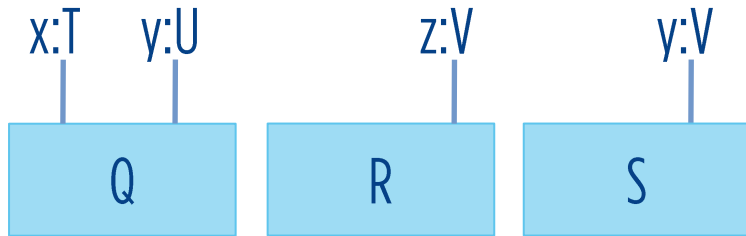
```

{ T, U, V, ... }
%%
{
...
(U,V,r),
(T,V,r1),
...
}
    
```

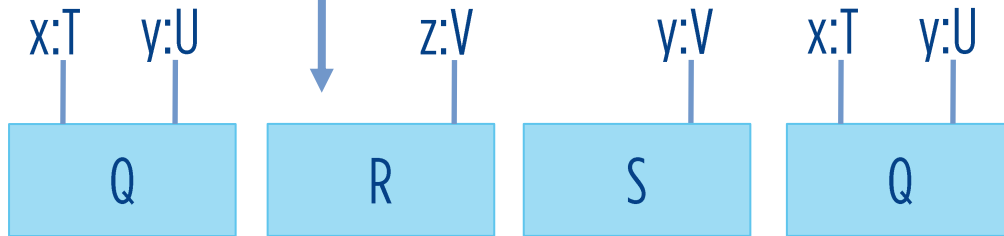
# BLENX: BIMOLECULAR ACTIONS



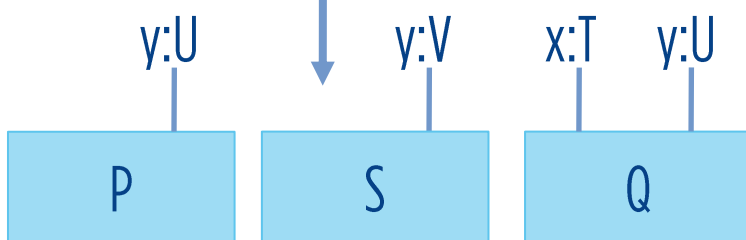
# BLENX: EVENTS



**when** ( Q :: 10 ) **new** ( 1 );

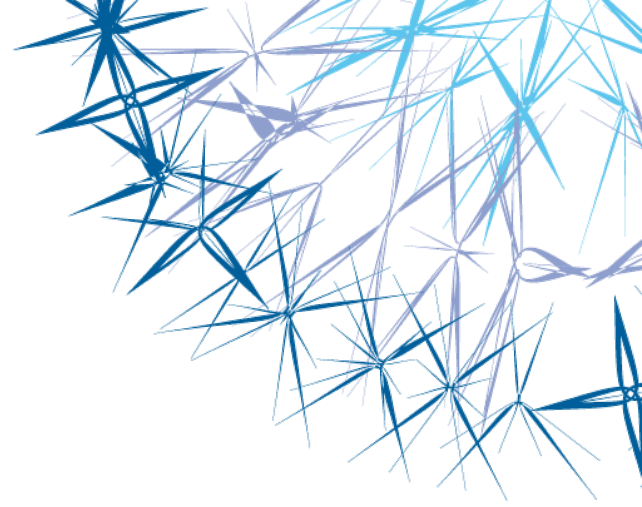


**when** ( R , Q :: f ) **join** ( P );  
**let** f = |S|\*sqrt(|Q|)/k

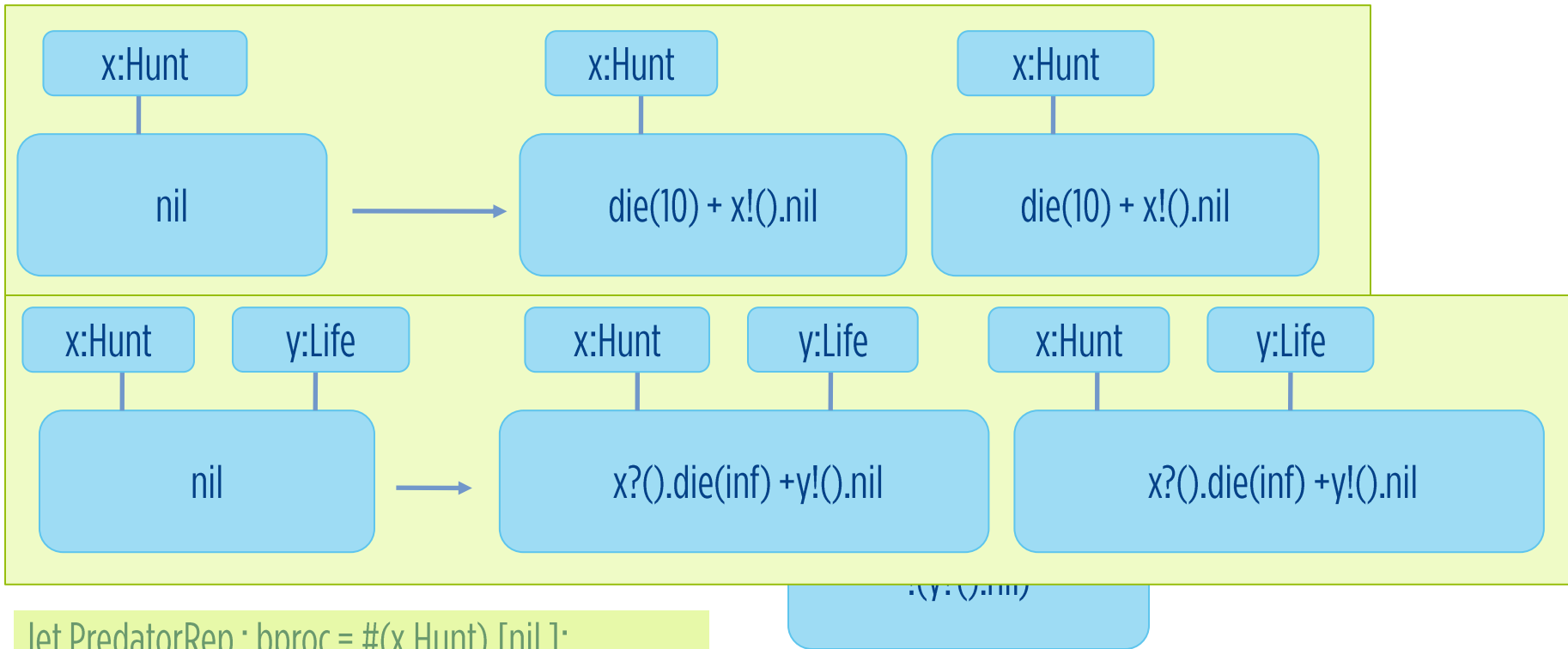


# Example

## Predator - Prey



# PREDATOR - PREY

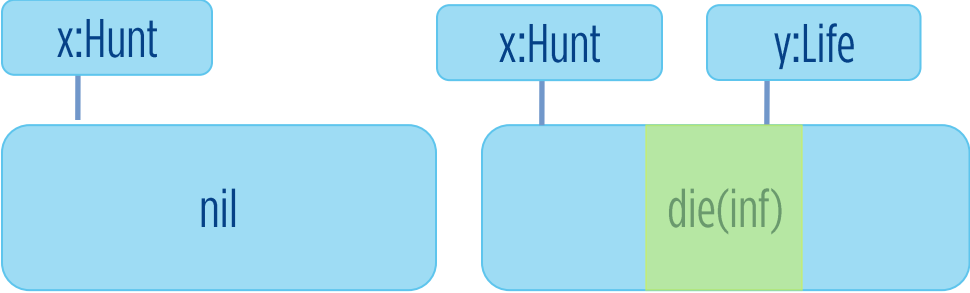
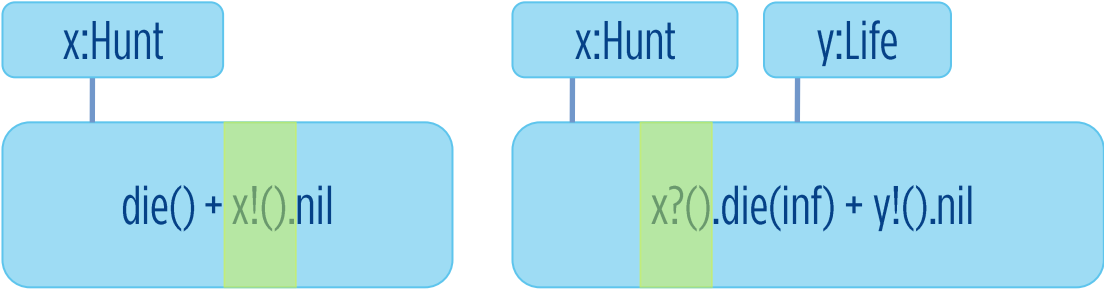


```
let PredatorRep : bproc = #(x,Hunt) [nil ];
when (PredatorRep :: inf) split (Predator,Predator);
```

```
let PreyRep : bproc = #(x,Hunt),#(y,Life) [nil ];
when (PreyRep :: inf) split (Prey,Prey) ;
```

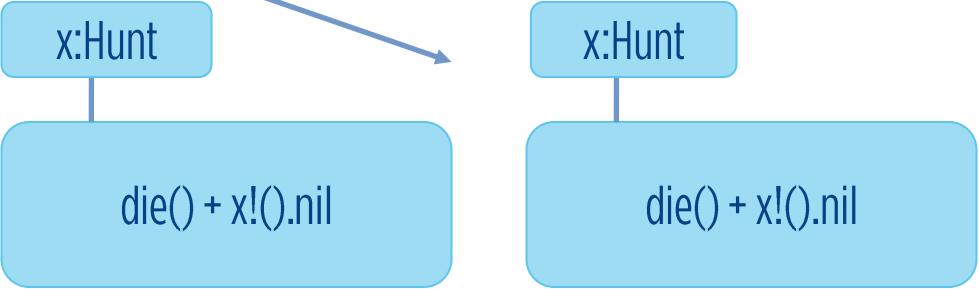
```
run 1000 Predator || 1000 Prey || 1 Nature
```

# PREDATOR - PREY



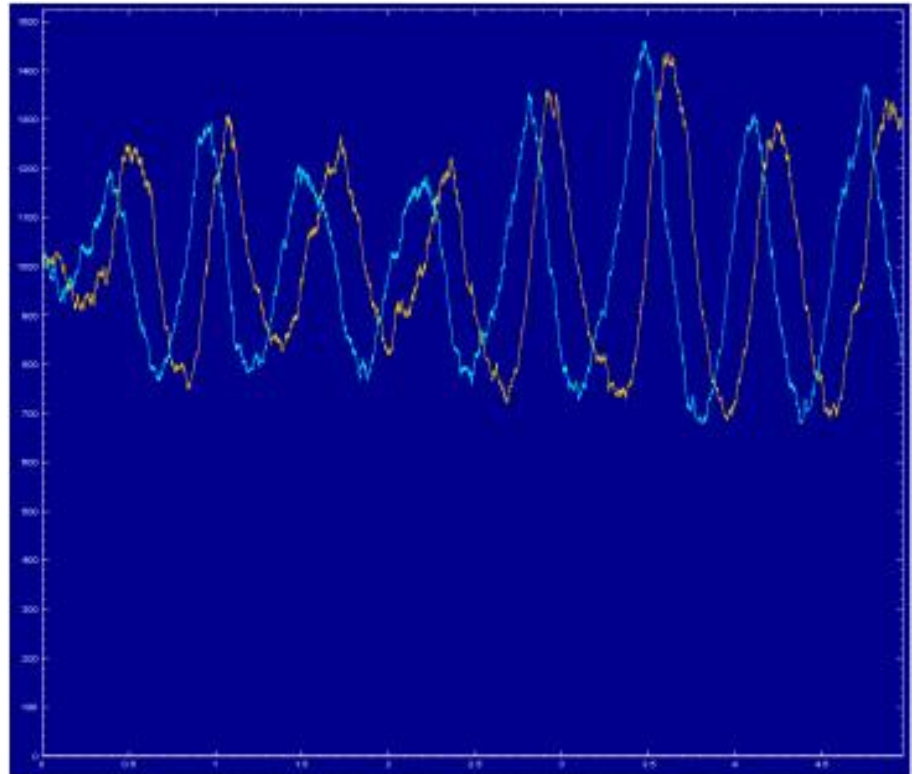
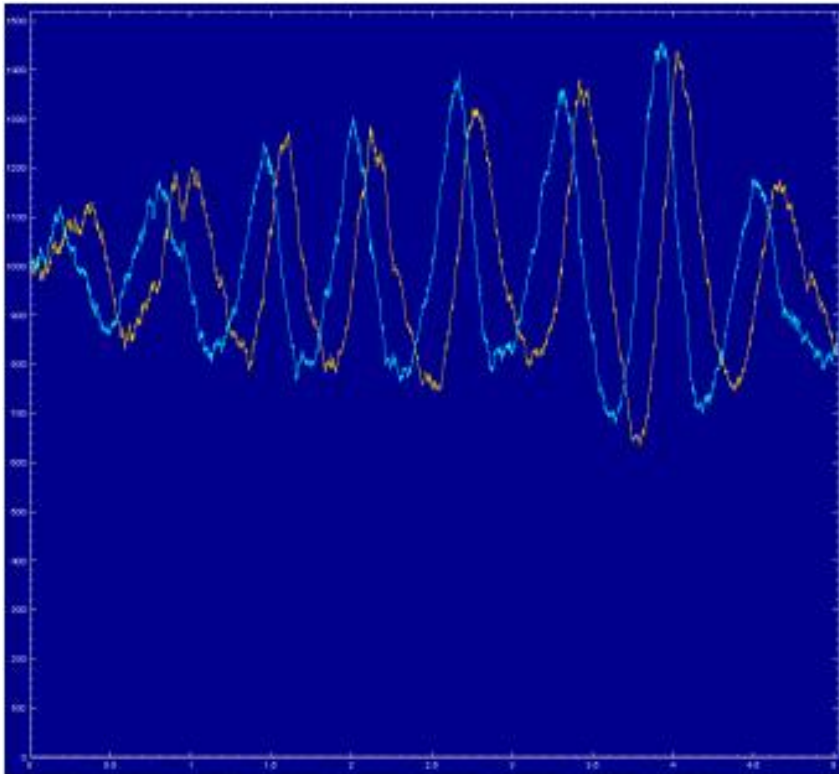
`{ Hunt, Life }`  
`%%`  
`{`  
`(Hunt,Hunt,10.0),`  
`(Life,Life,0.01)`  
`}`

When (PredatorRep::inf) split (Predator,Predator)



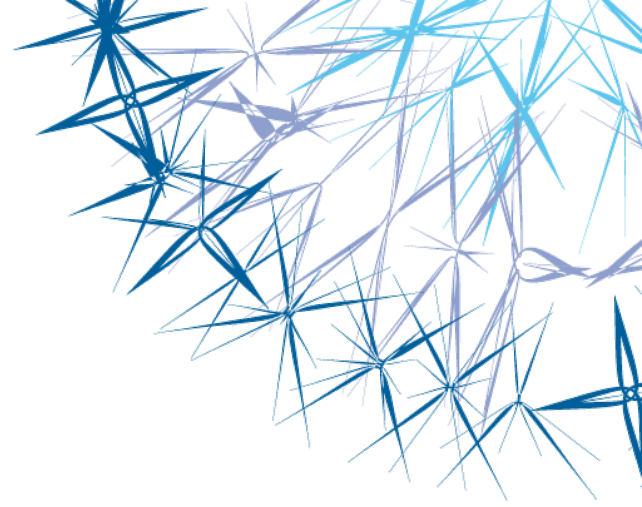
# PREDATOR - PREY

---

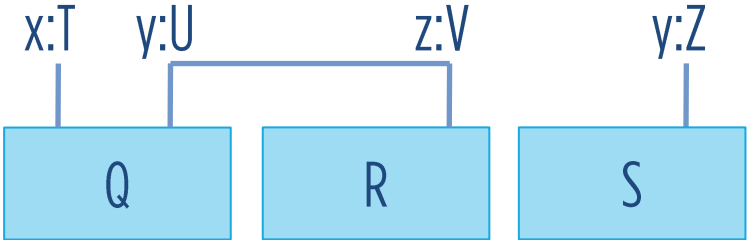
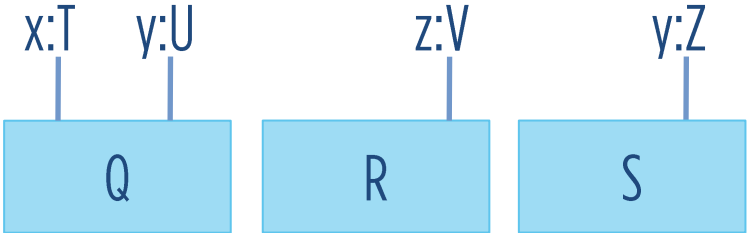


# END Example

## Predator - Prey



# BLENX: COMPLEXES



{T, U, V, Z, ... }

%%

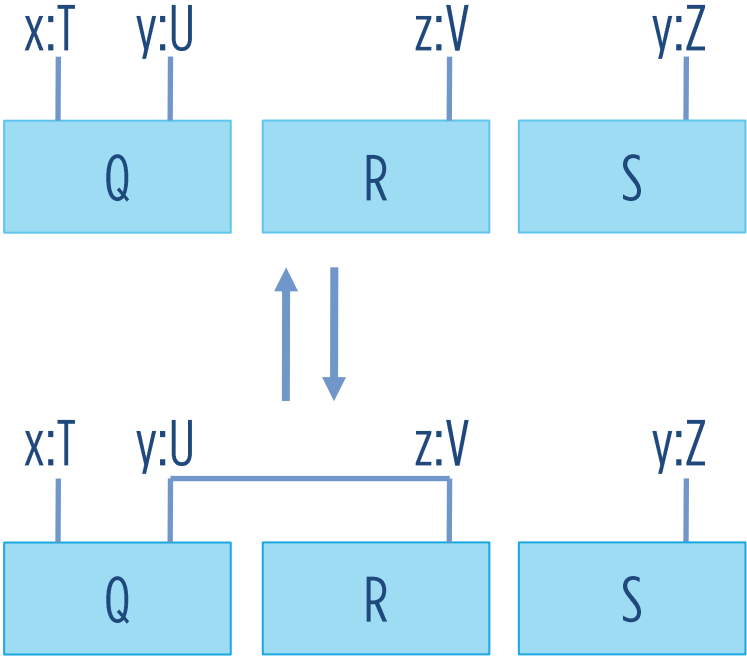
{

(U, Z, 3.5),

(U, V, 1.5)

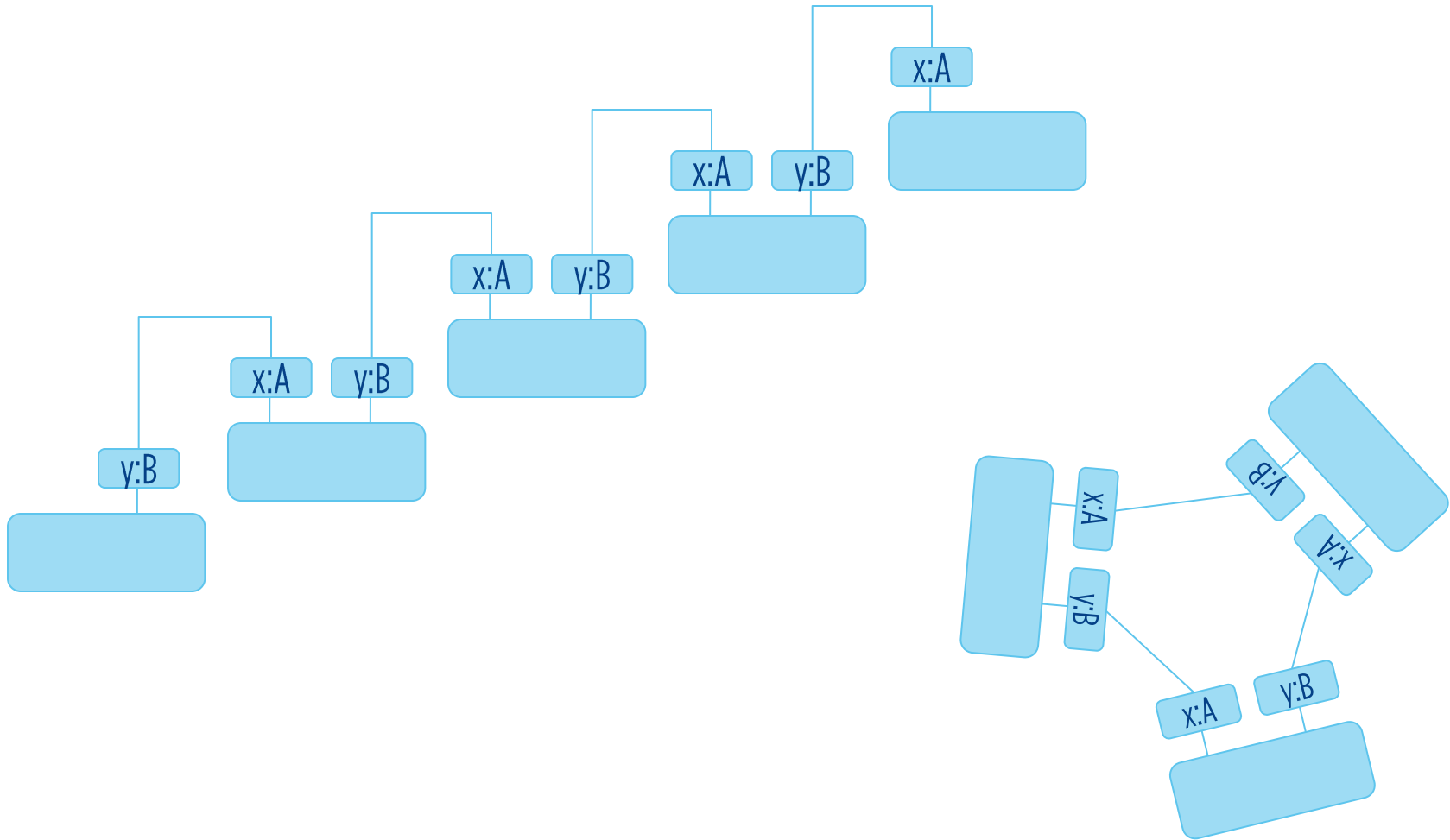
}

# BLENX: COMPLEXES



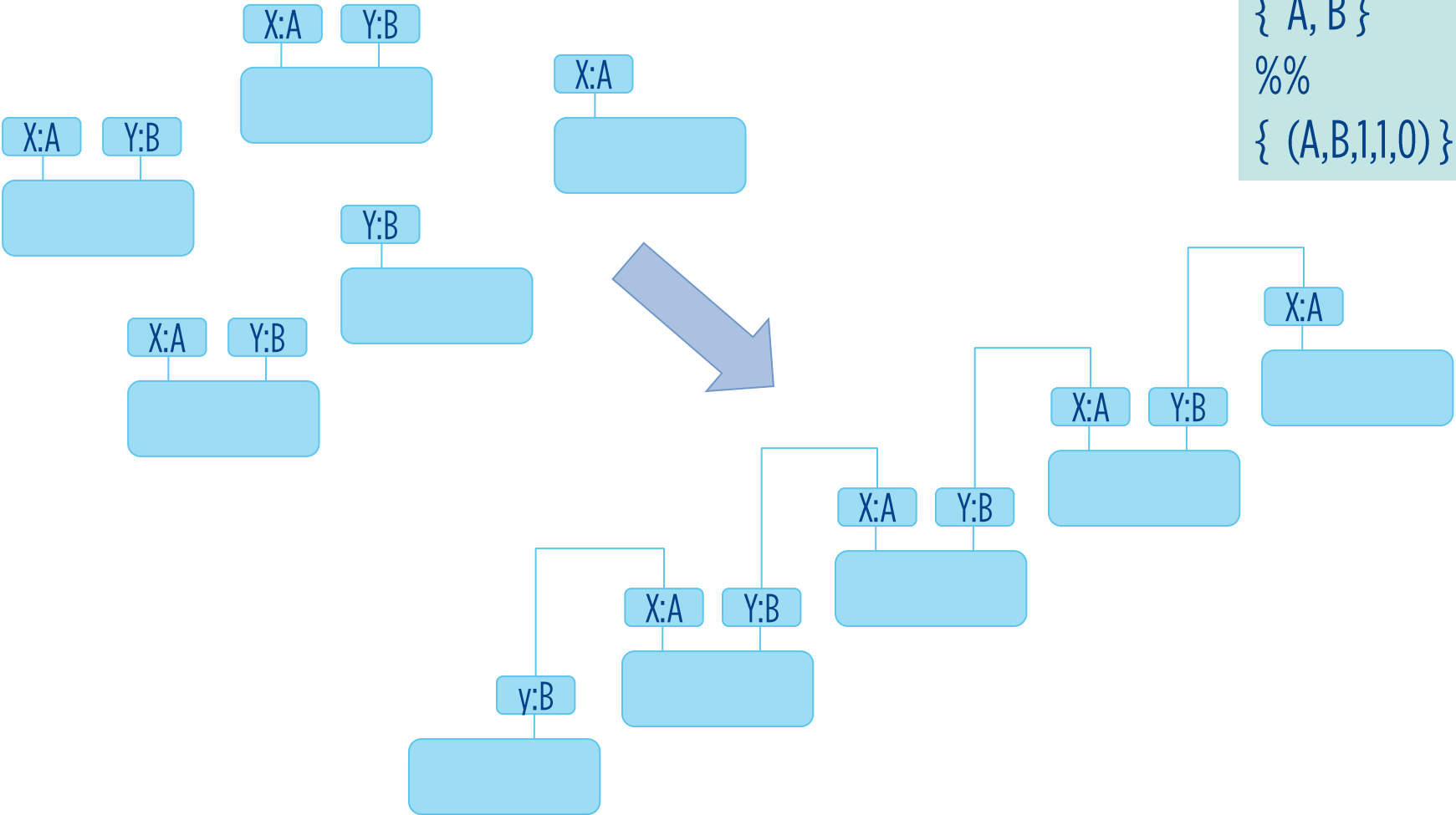
{T, U, V, Z, ... }  
%%  
{  
(U, Z, 3.5),  
(U, V, 1.5, **2.5, 10**)  
}

# BLENX: STRUCTURES

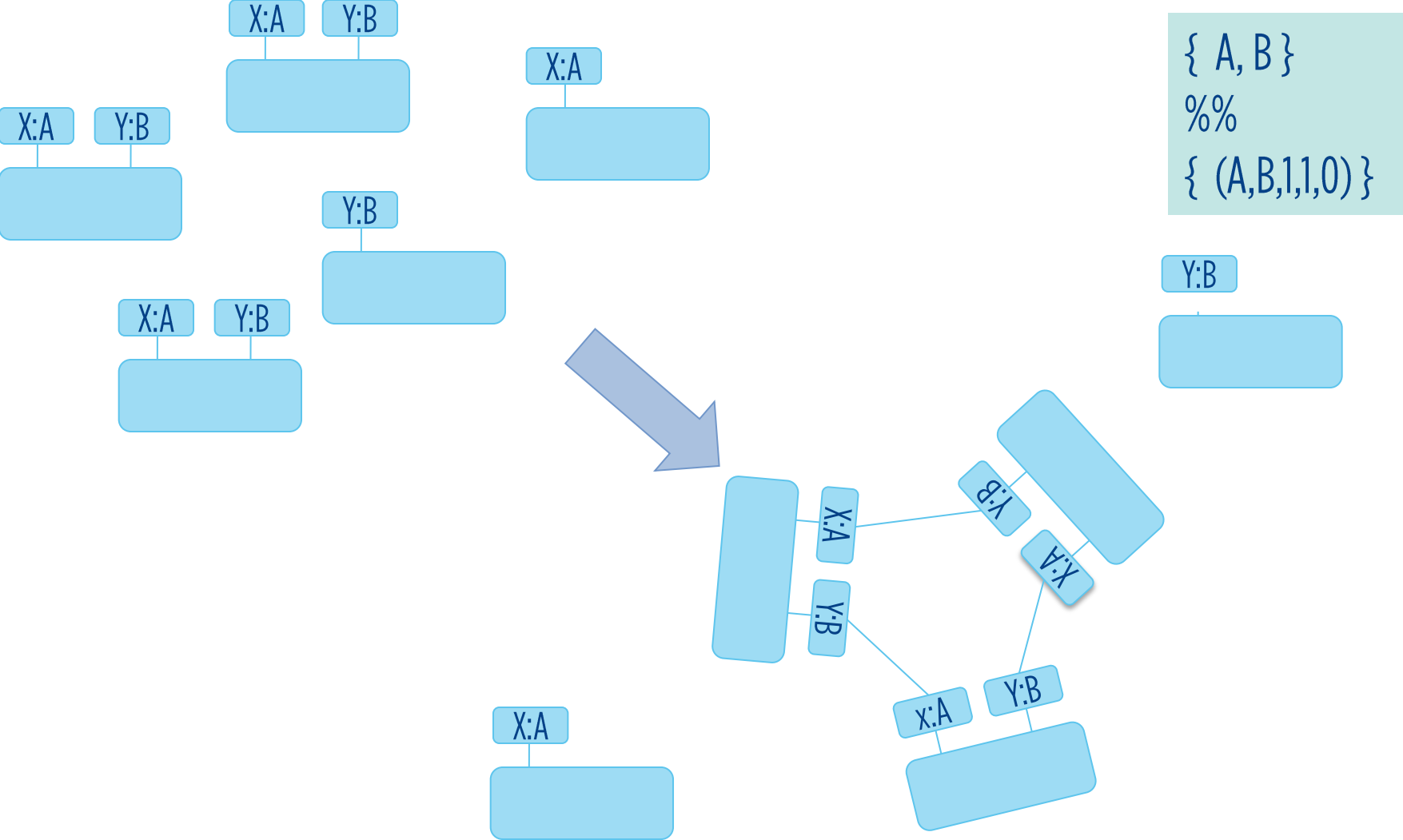


R. Larcher, C. Priami and A. Romanel. *Modelling self-assembly in BlenX*.  
Transactions on Computational Systems Biology, XII:LNBI 5945,  
163-198, 2010.

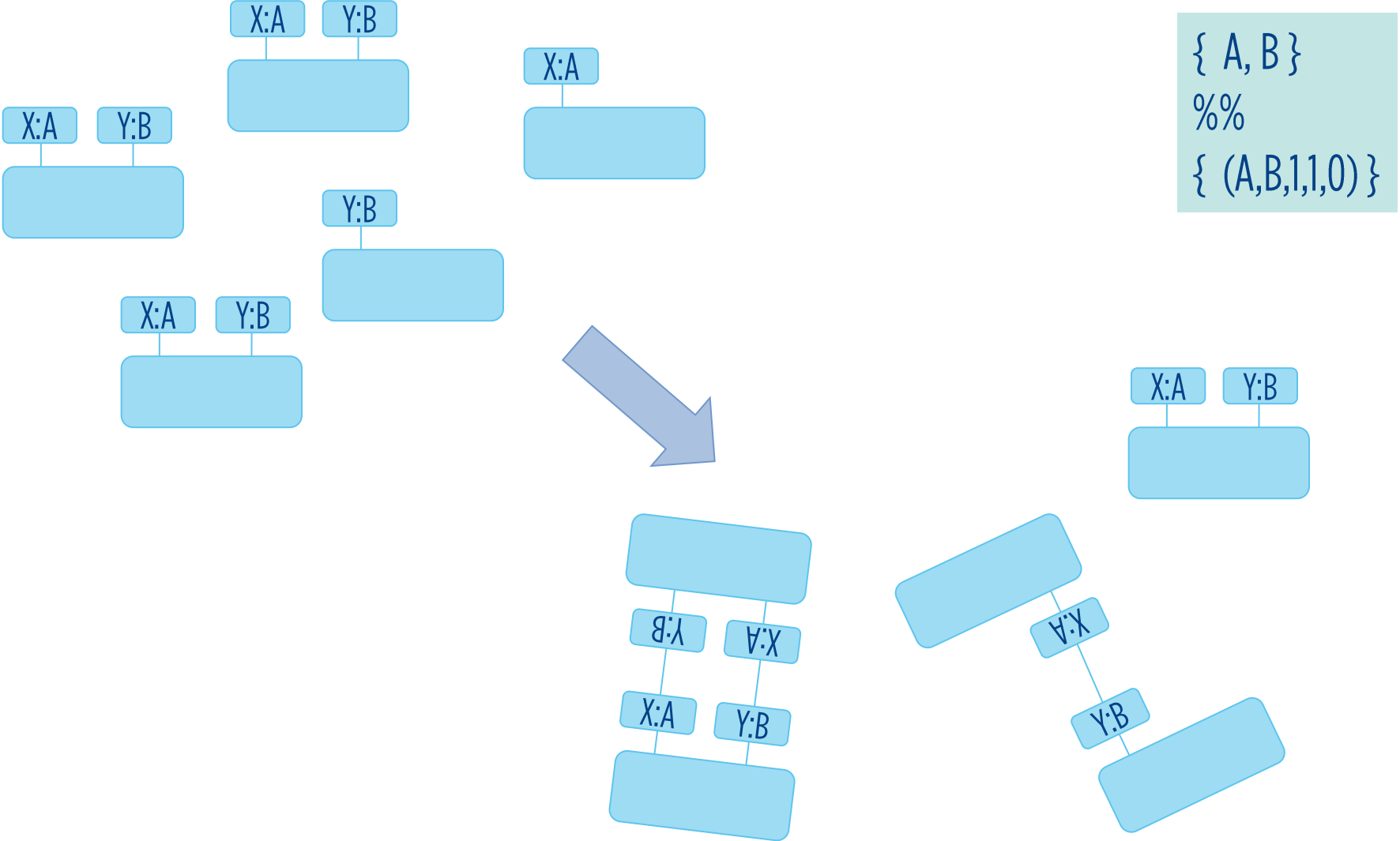
# DYNAMICALLY CREATING STRUCTURES



# DYNAMICALLY CREATING STRUCTURES



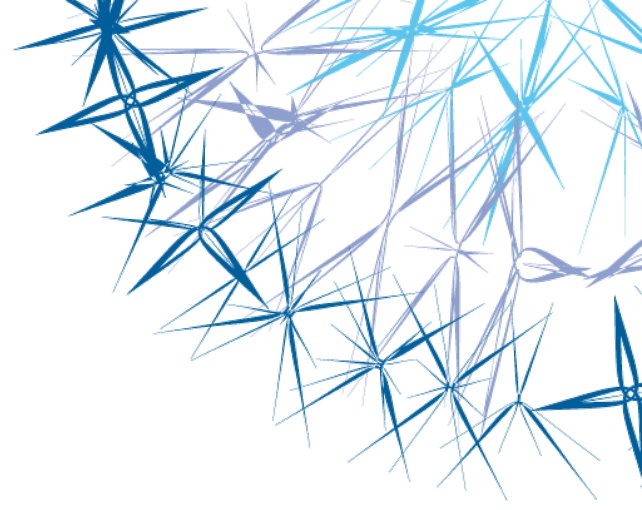
# DYNAMICALLY CREATING STRUCTURES



# Example

## SELF assembly of trees

---



# SELF ASSEMBLY OF TREES

```
// Tree.prog

[ steps = 10, delta = 10 ]

<< BASERATE:inf, HIDE:inf, UNHIDE:inf >>

// Initiator Definition
let Initiator : bproc = #(out,I) [ out?().out!(root).nil ];

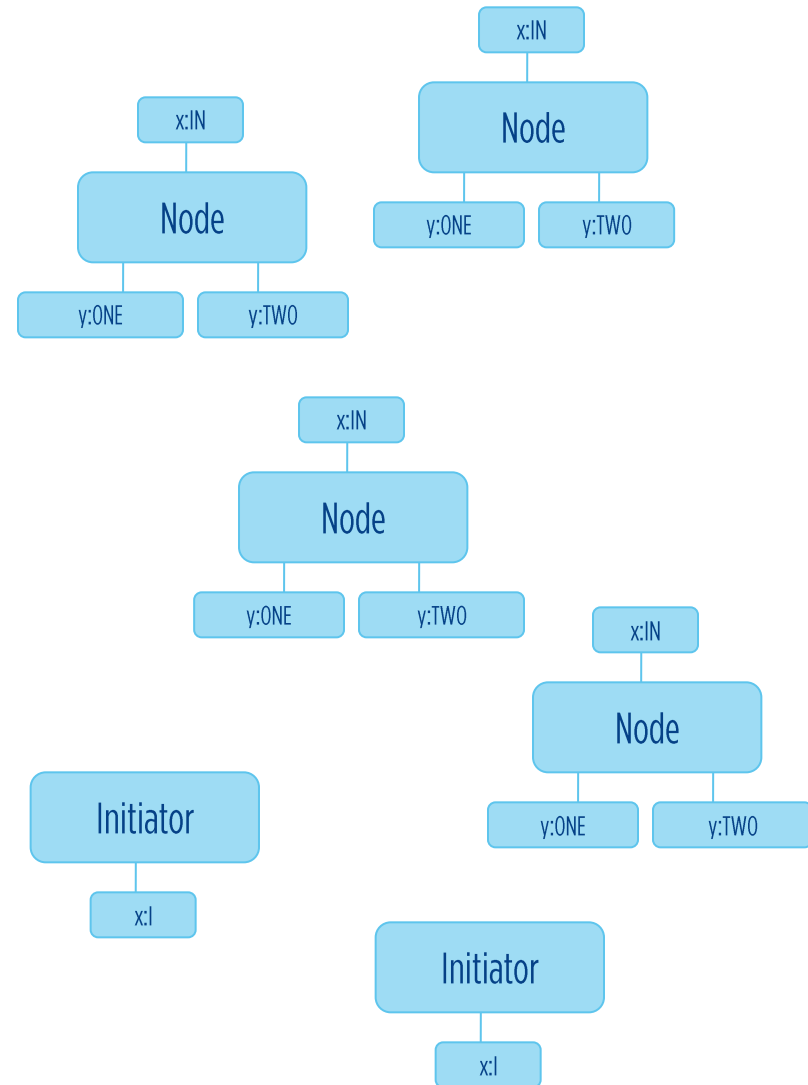
// Node Definition
let p1 : pproc = !(out1?().out2?().out1!(node).out2!(node).nil) ;
let p2 : pproc = !(out1?().out2?().inp!().inp?(m).out1!(m).out2!(m).nil) ;
let nodeP : pproc =
  inp!().inp?(t).( t!() | (
    node?().unhide(out1).unhide(out2).p2 +
    root?().unhide(out1).unhide(out2).p1
  ) );

let Node : bproc = #(inp,IN),#h(out1,ONE),#h(out2,TWO)
  [ nodeP ];

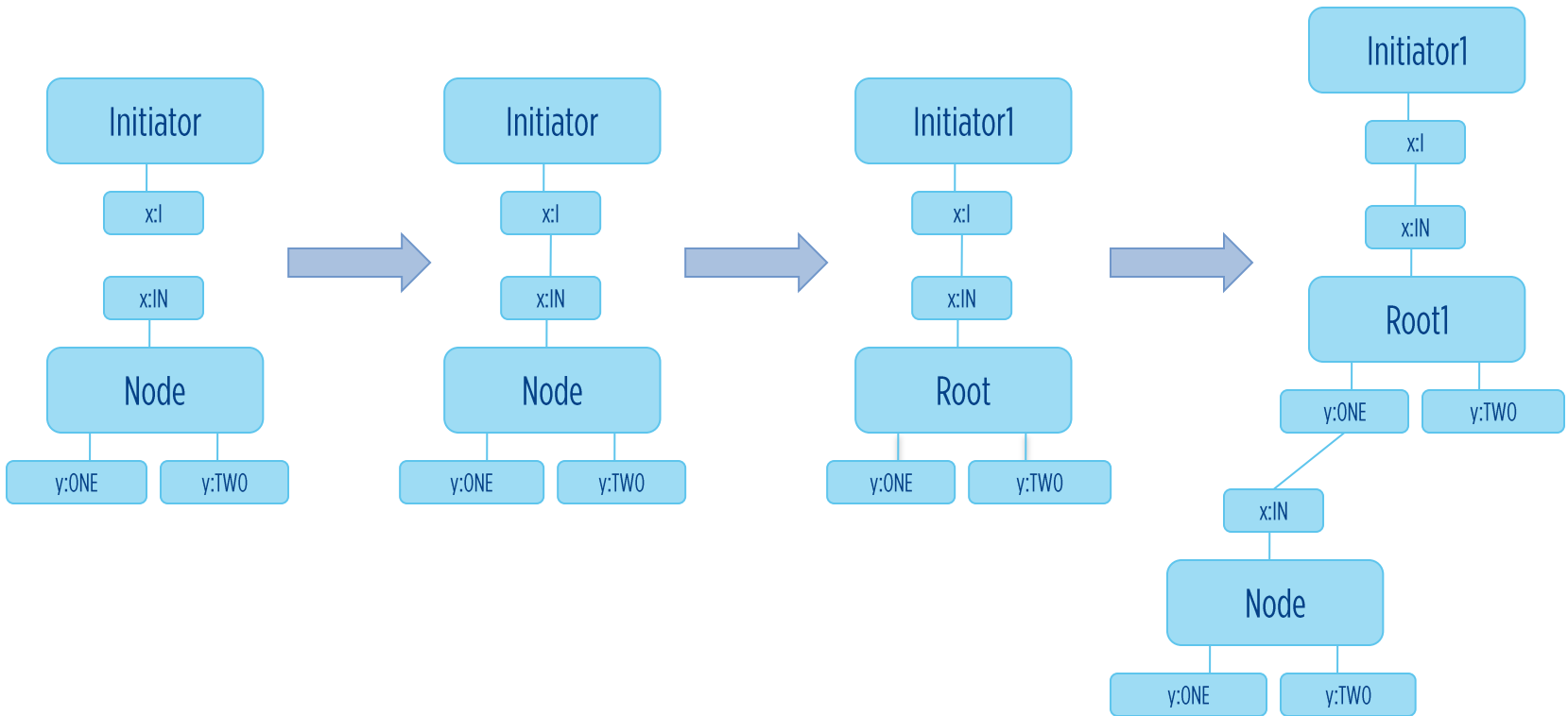
// Init
run 2 Initiator || 10 Node
```

```
// Tree.types
```

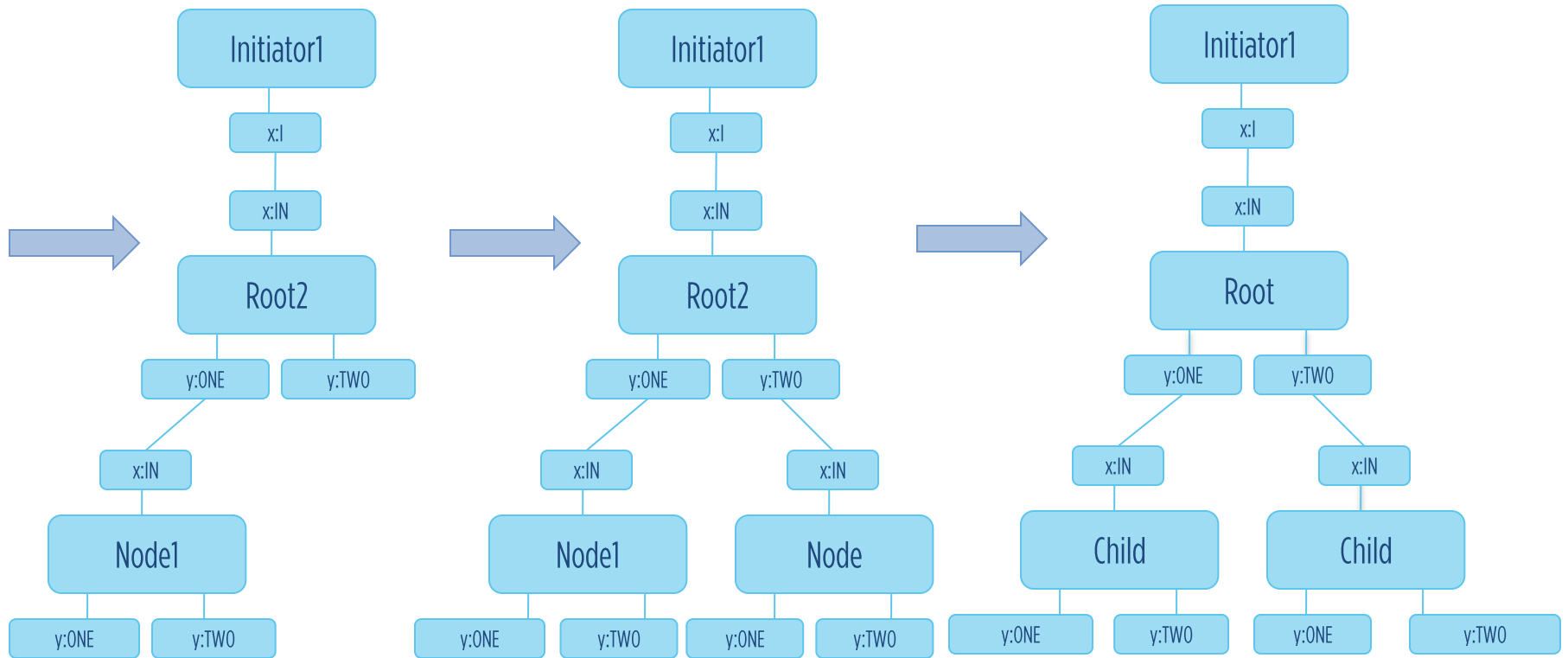
```
{ I, IN, ONE, TWO }
%%
{
  (I,IN,100,0,inf),
  (ONE,IN,1,0,inf),
  (TWO,IN,1,0,inf)
}
```



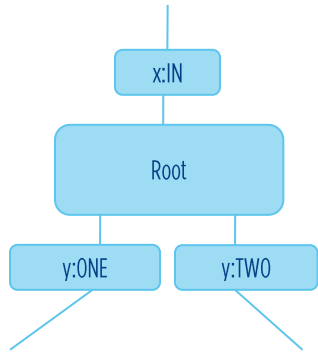
# SELF ASSEMBLY OF TREES



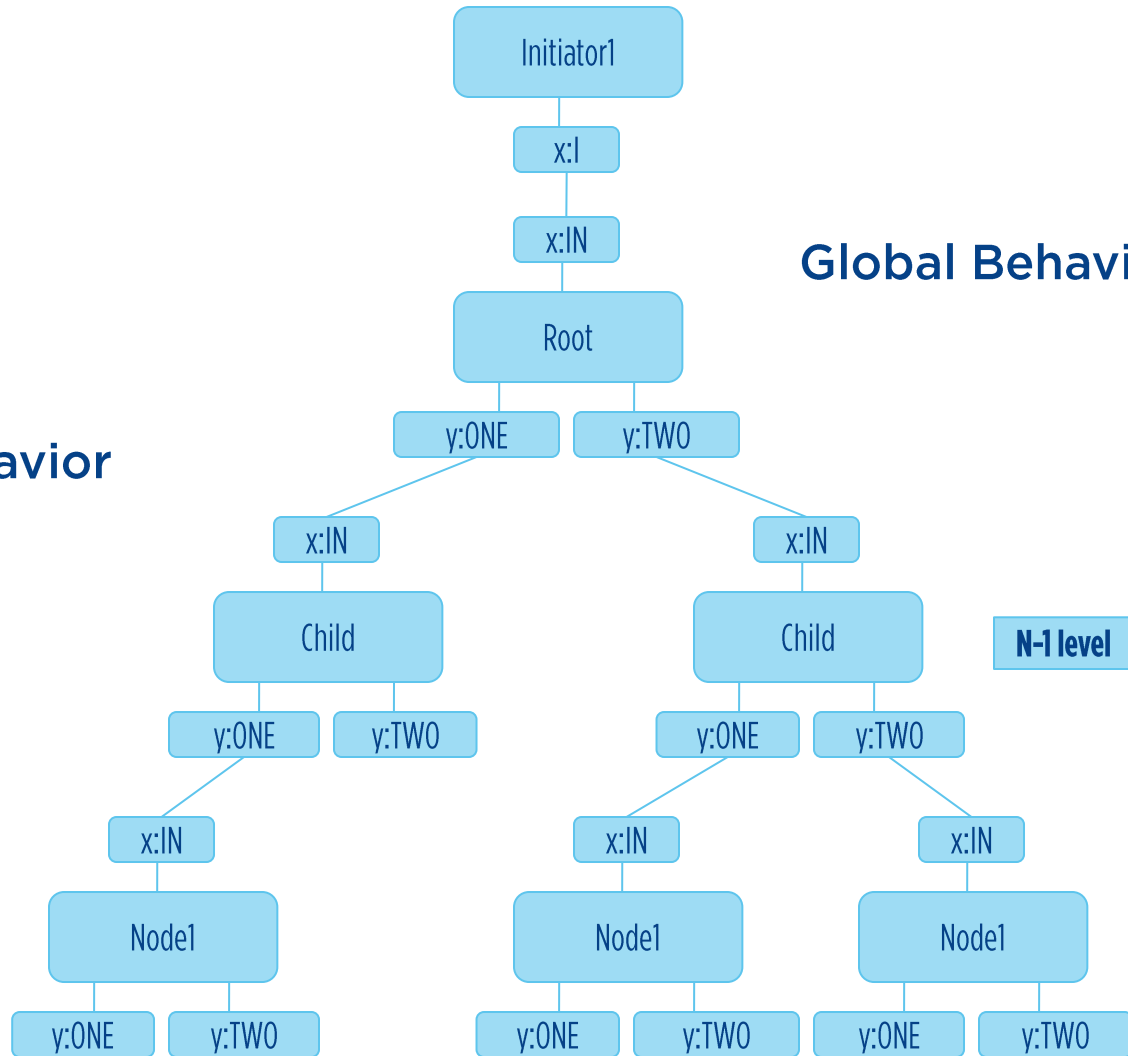
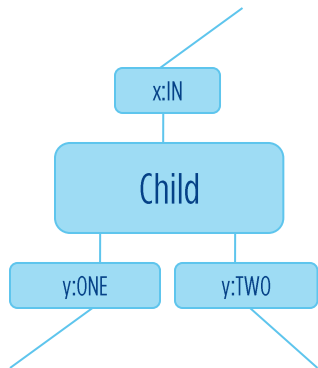
# SELF ASSEMBLY OF TREES



# SELF ASSEMBLY OF TREES

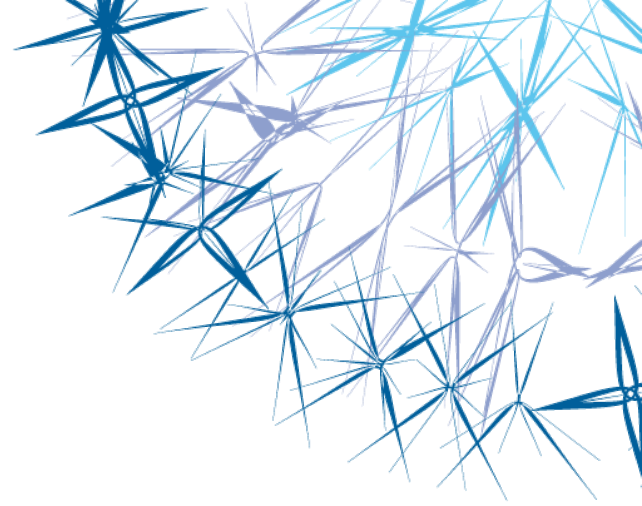


Local Behavior



Global Behavior

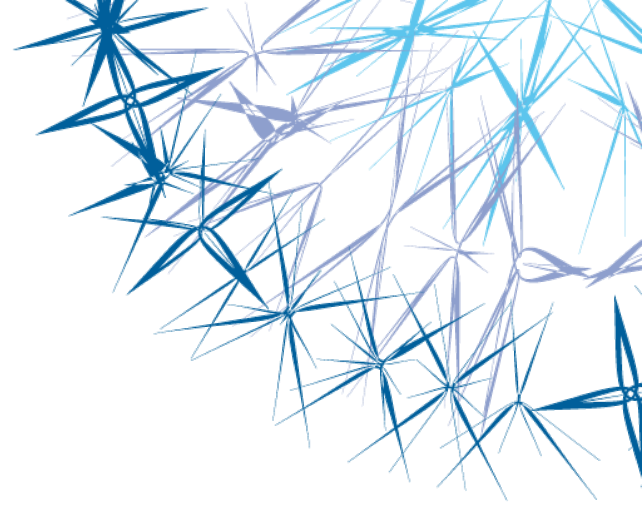
END Example  
SELF assembly of trees



# BLENX

## RECAP MAIN CONCEPTS

---



# RECAP BLENX

---

- Simulation and analysis (causality, logical properties)
- Scalability, modularity, compositionality
- Different levels of abstraction and refinement
- Easy models, easy libraries due to combinatorial effects ruled out at modeling level
- Executable vs. solvable specifications, modules vs. variables, dynamic relations vs. static relations