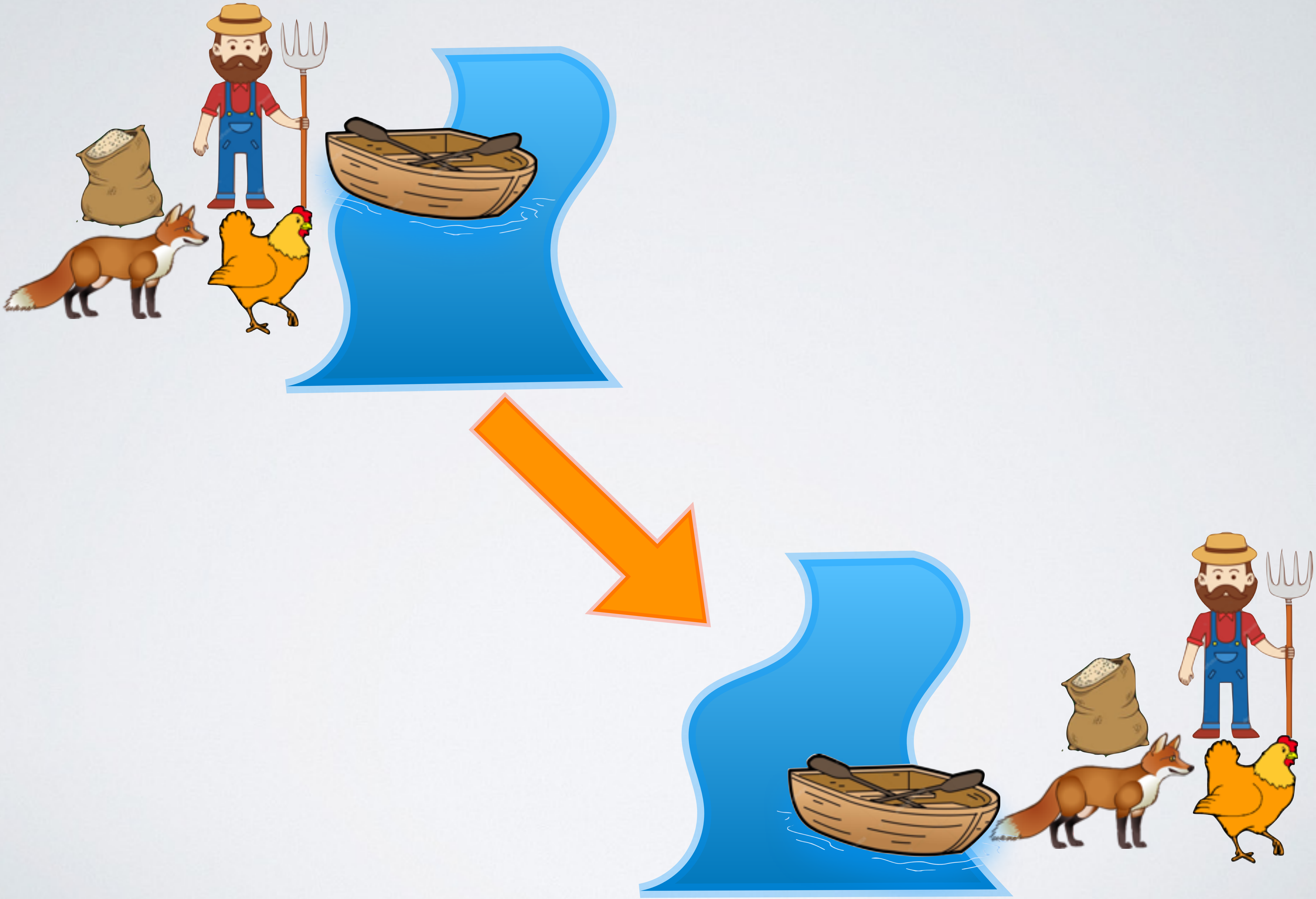
DYNALLOY: AN EXTENSION OF ALLOY FOR WRITING AND ANALYZING BEHAVIOURAL MODELS

Germán Regis | César Cornejo | Simón Gutiérrez Brida | Mariano Politano | Fernando Raverta | Pablo Ponzio | Nazareno Aguirre | Juan Pablo Galeotti | Marcelo Frias

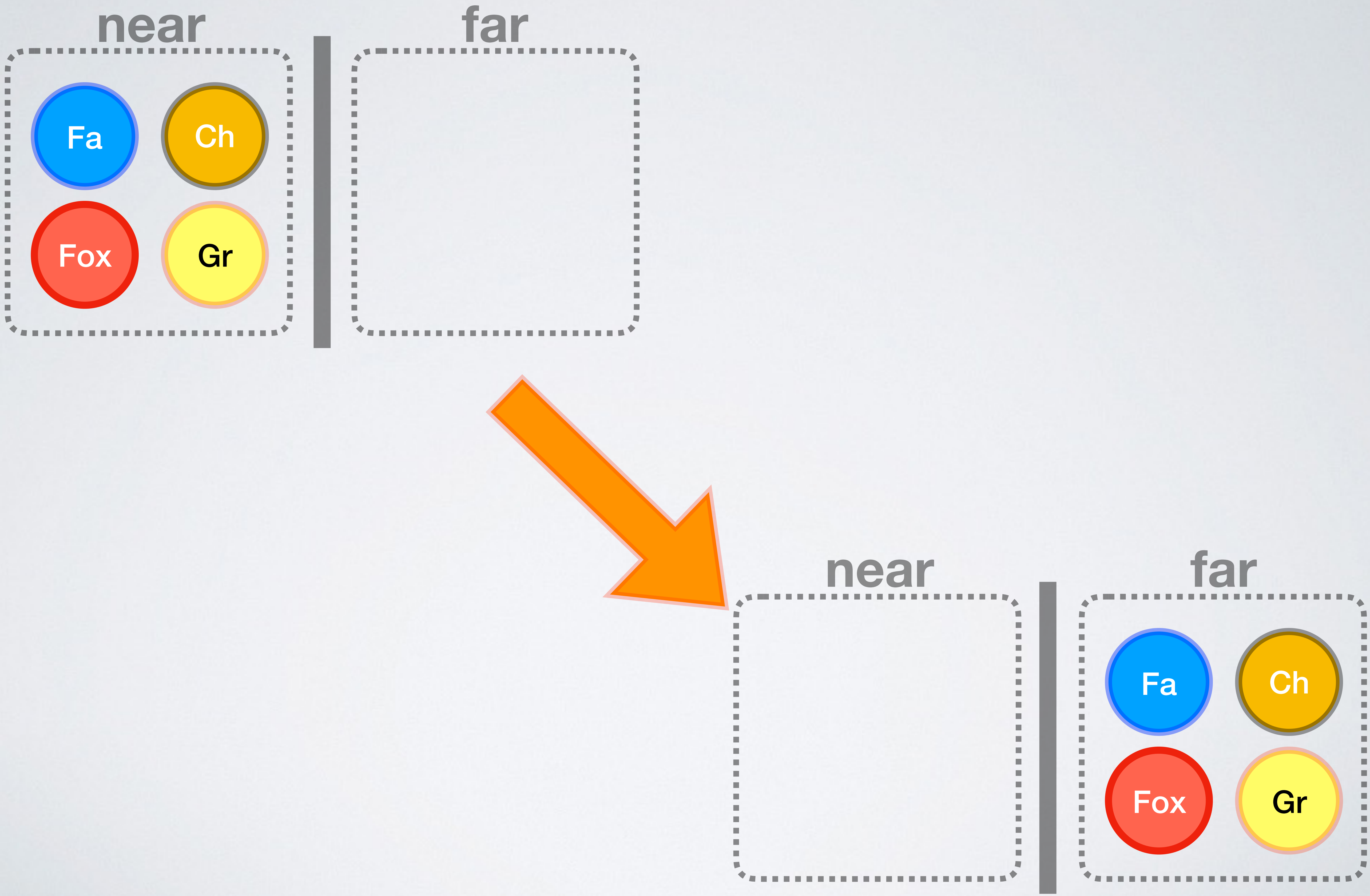
Universidad Nacional de Río Cuarto
Universidad de Buenos Aires
Instituto Tecnológico Buenos Aires

Workshop on the Future of Alloy

EXAMPLE - RIVER CROSSING PUZZLE

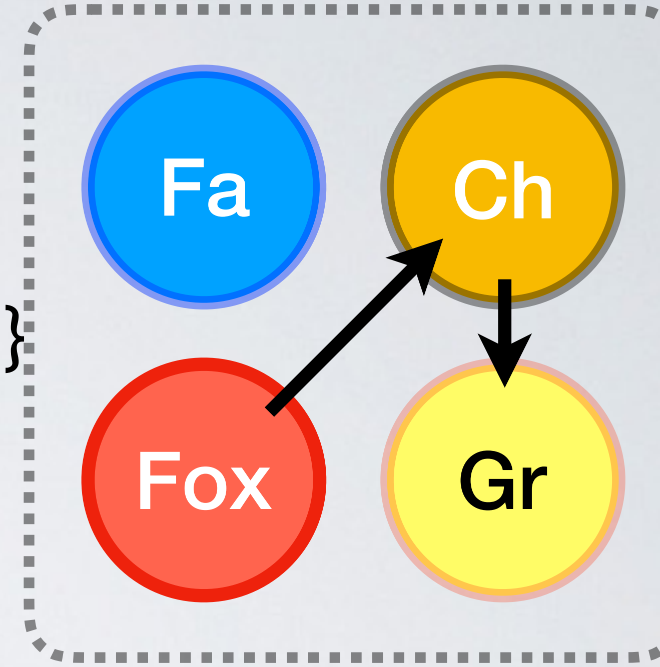


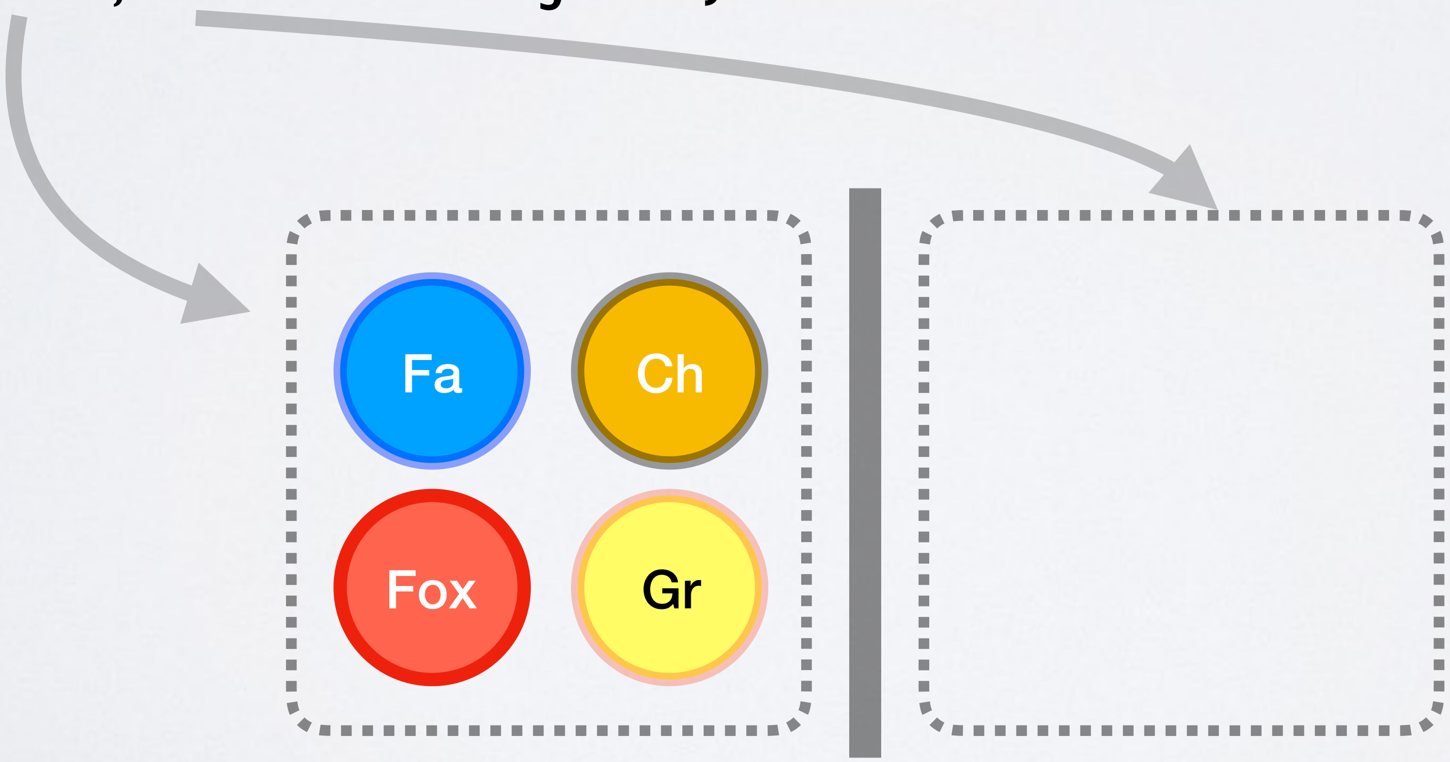
EXAMPLE - RIVER CROSSING PUZZLE



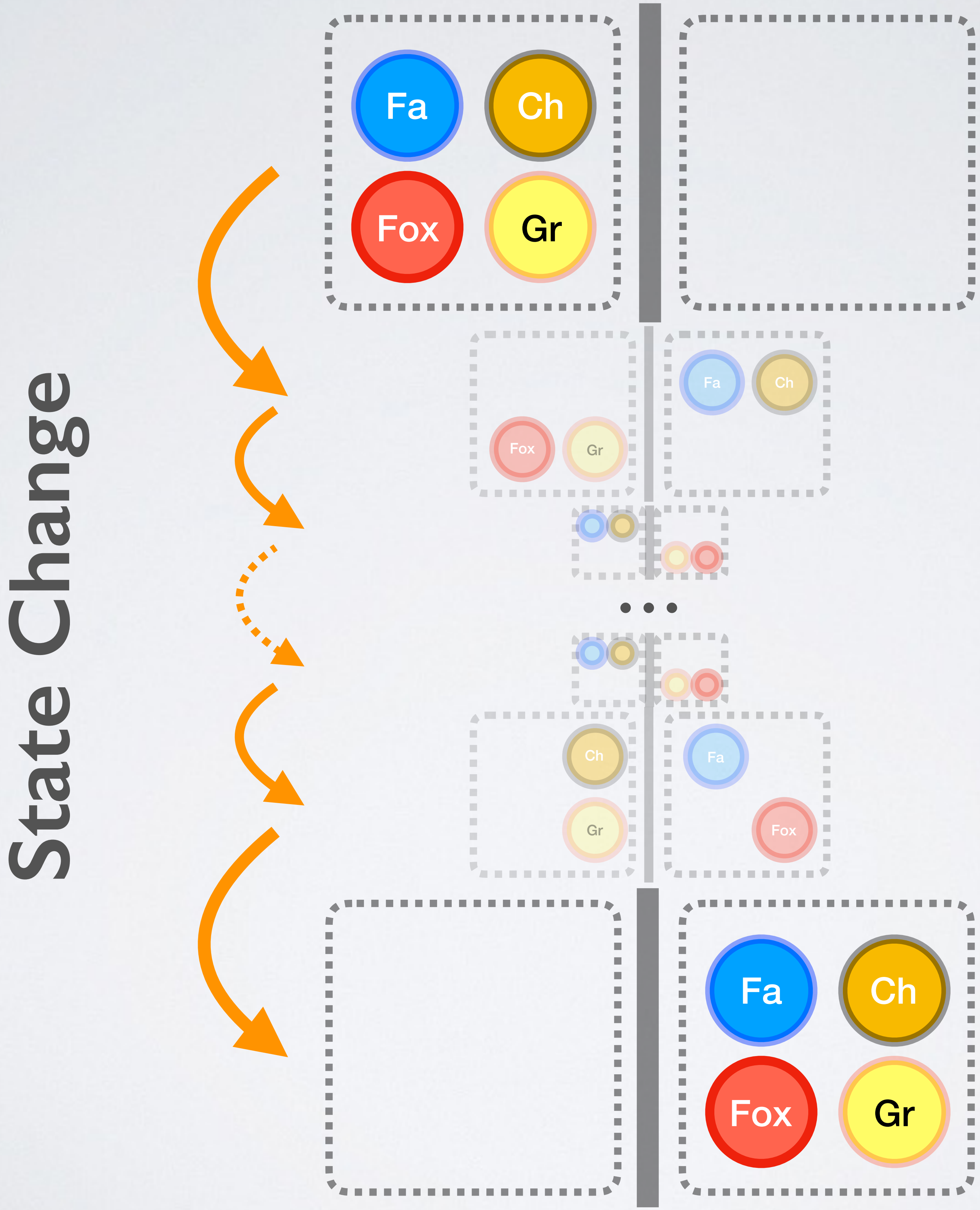
RIVER CROSS - ALLOY SPECIFICATION

```
abstract sig Object { eats: set Object}

one sig Farmer, Fox, Chicken, Grain extends Object { }

fact { eats = Fox->Chicken + Chicken->Grain }

sig State { near, far: set Object }
```

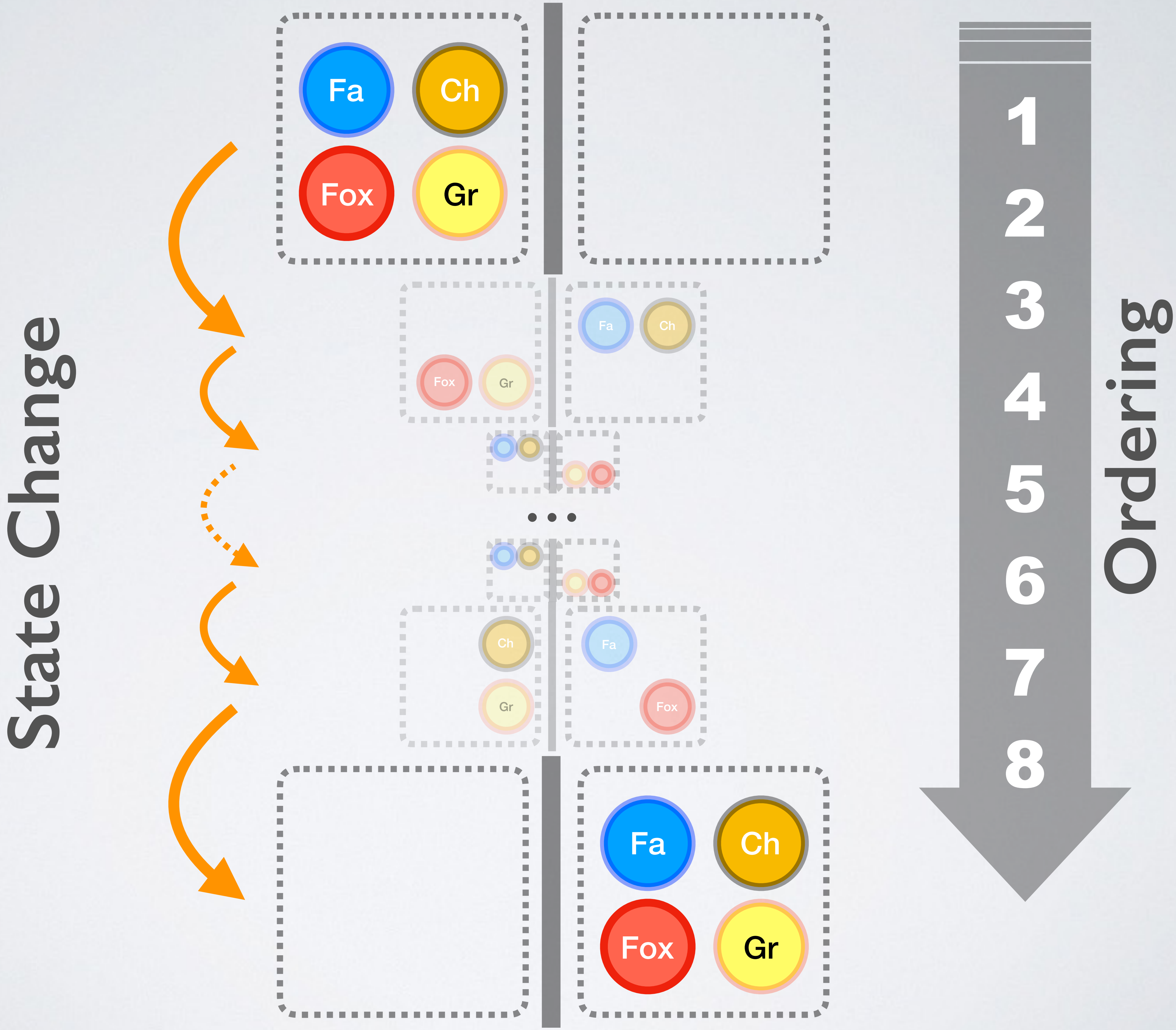




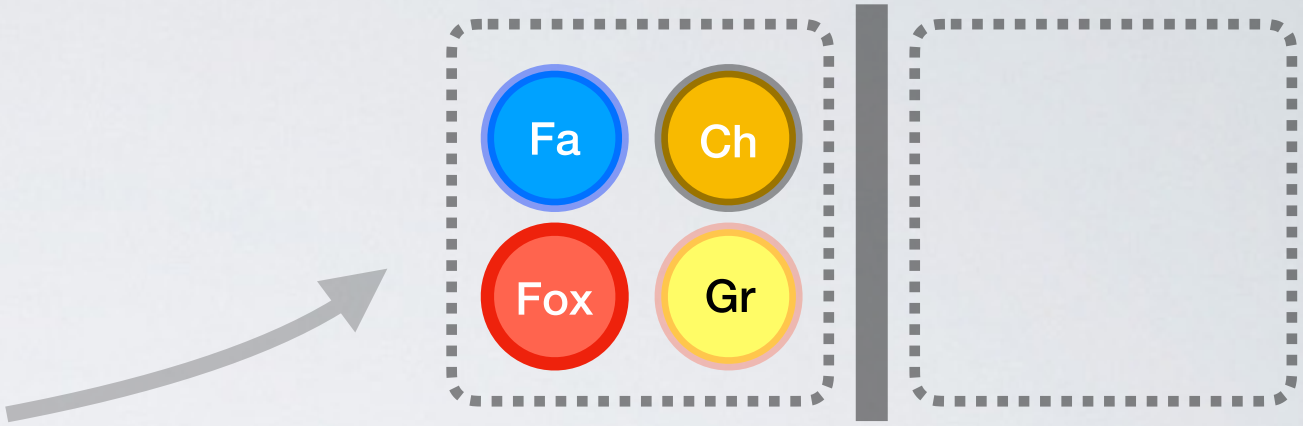
RIVER CROSS - DYNAMIC BEHAVIOR



RIVER CROSS - DYNAMIC BEHAVIOR

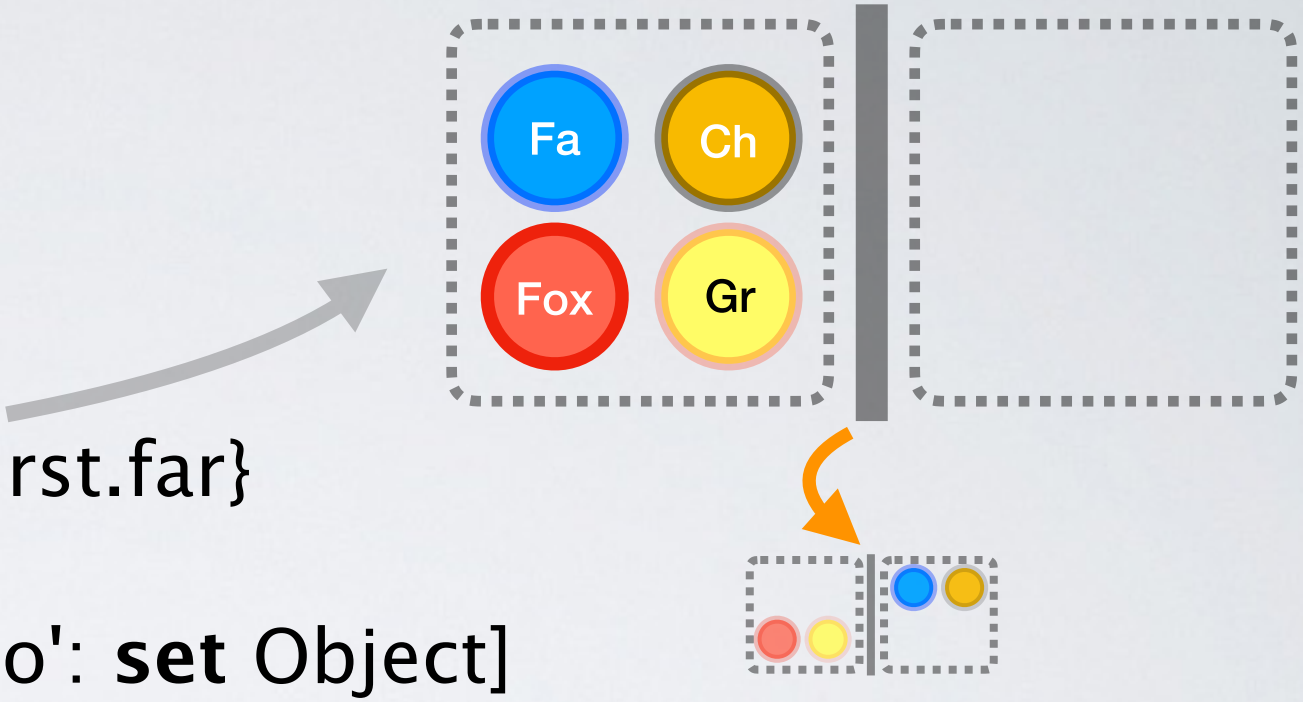


RIVER CROSS - ALLOY SPECIFICATION



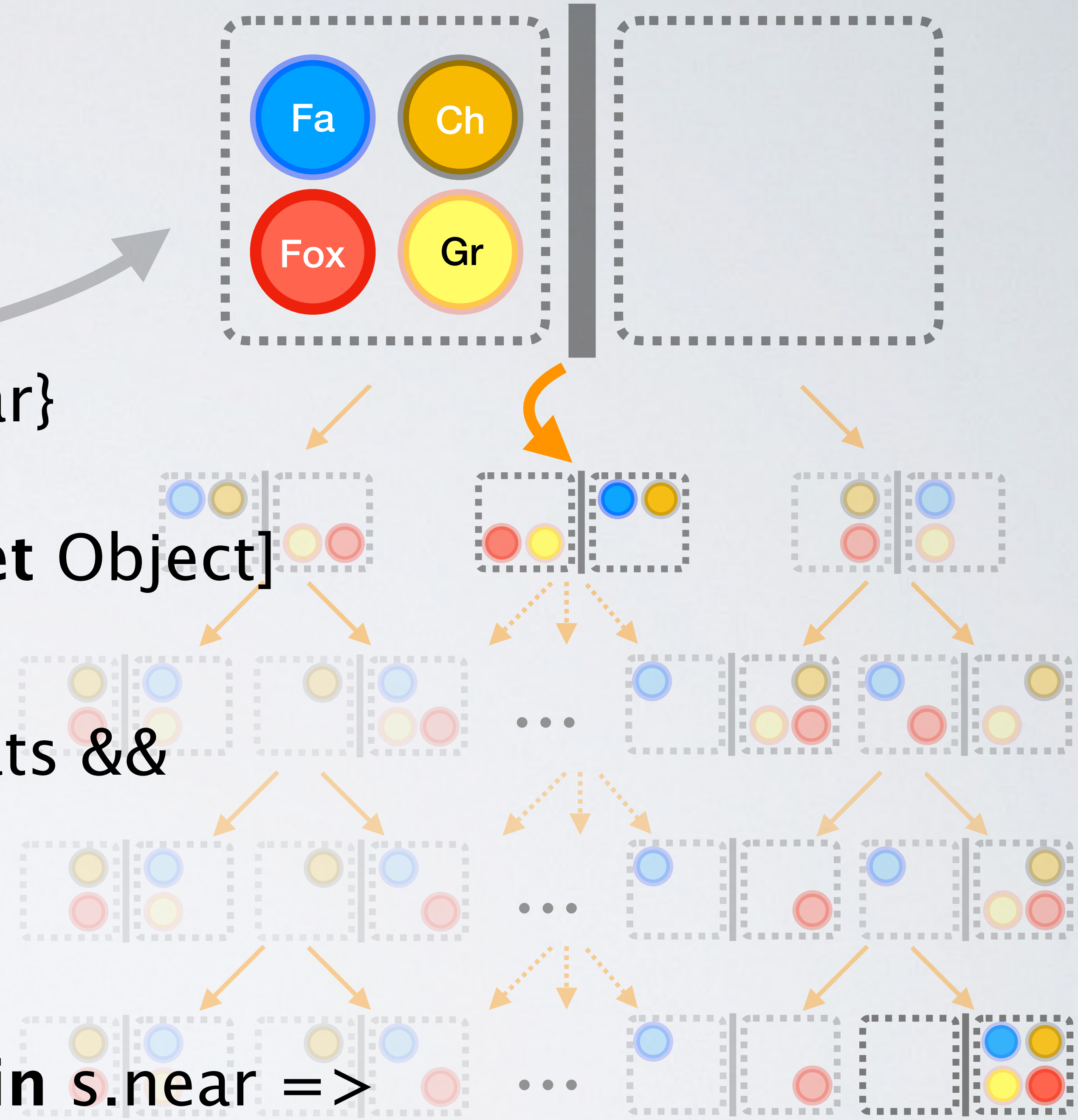

```
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}
```

RIVER CROSS - ALLOY SPECIFICATION




```
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
  from' = from – x – Farmer – from’.eats &&
  to' = to + x + Farmer }
}
```

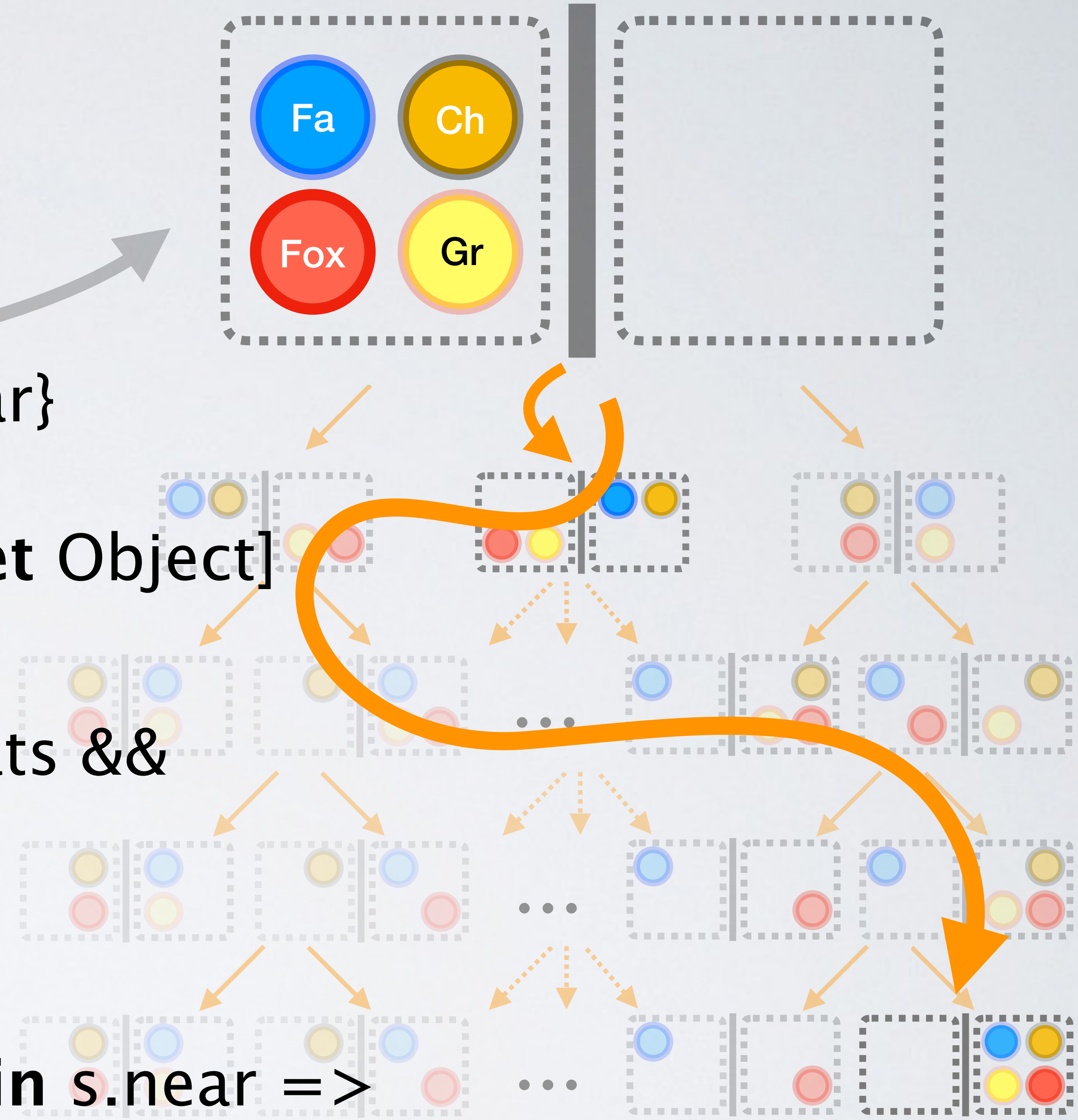
RIVER CROSS - ALLOY SPECIFICATION




```
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
  from' = from – x – Farmer – from’.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}
```

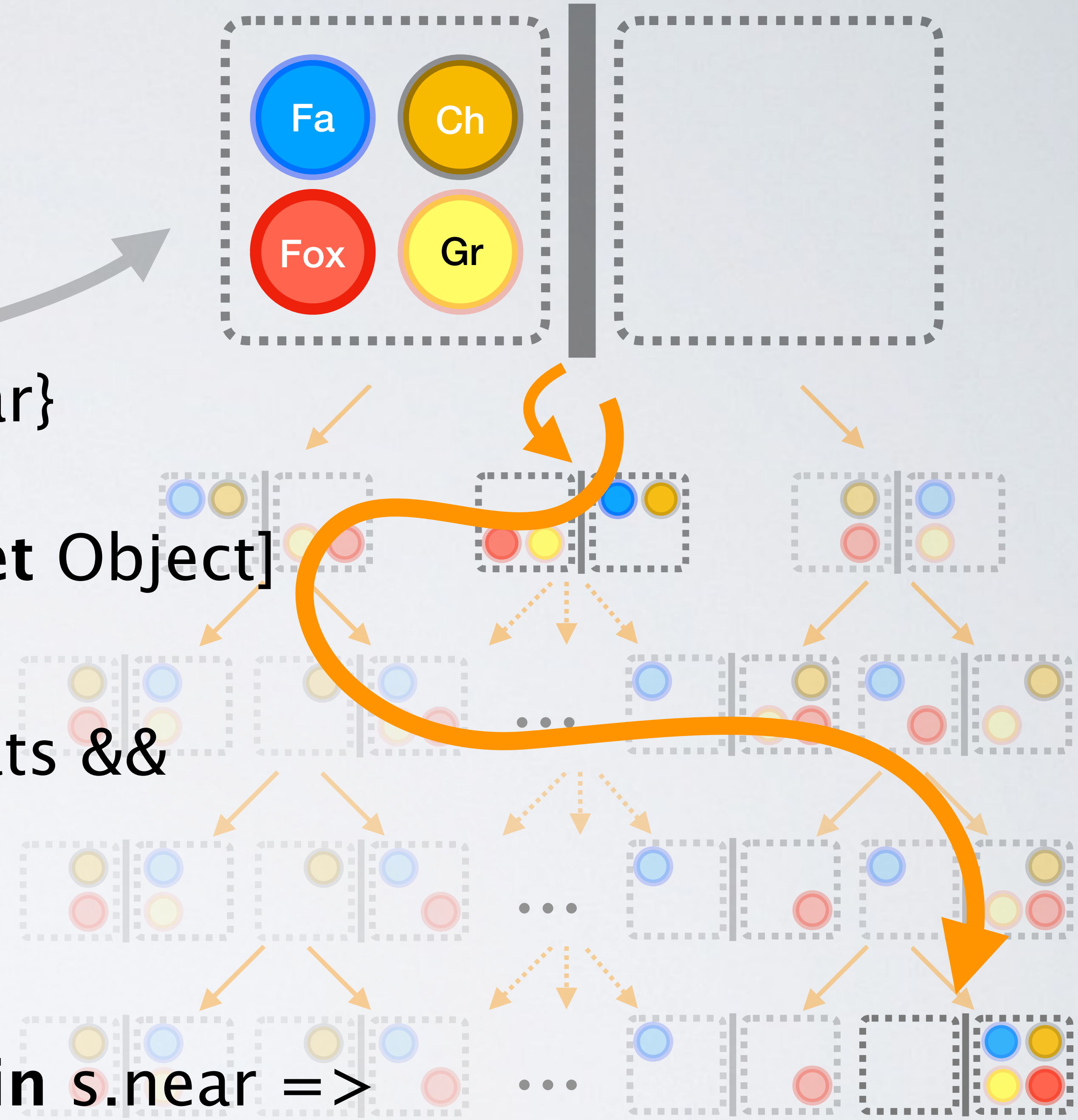
RIVER CROSS - ALLOY SPECIFICATION

```
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
  from' = from – x – Farmer – from’.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States
```



SATisfying valuations of the predicate are solutions to the puzzle

RIVER CROSS - ALLOY SPECIFICATION




```
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
  from' = from – x – Farmer – from'.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States
```

SATisfying valuations of the predicate are solutions to the puzzle

DYNALLOY
=
ALLOY + DYNAMIC LOGIC

- Execution traces are indirectly defined through:
  - **Atomic Actions** (State Change)
  - **Programs** (imperative style & nondeterminism)

    Assumptions | Test **?** | Choice **+** |

    Sequential Composition **;** | Iteration *****

RIVER CROSS - DYNALLOY SPECIFICATION

```
open util/ordering[State]
sig State { near, far: set Object }
fact { first.near = Object && no first.far}

pred   crossRiver[from,from', to , to' : set Object]
{ one x: from | {
  from' = from – x – Farmer – from'.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States
```

RIVER CROSS - DYNALLOY SPECIFICATION

```
sig State { near, far: set Object }
fact { first.near = Object && no first.far}
pred   crossRiver[from,from’, to , to’ : set Object]
{ one x: from | {
  from' = from – x – Farmer – from’.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States
```

RIVER CROSS - DYNALLOY SPECIFICATION

```
sig State { near, far: set Object }

pred   crossRiver[from,from’, to , to’ : set Object]
{ one x: from | {
  from' = from – x – Farmer – from’.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States
```

RIVER CROSS - DYNALLOY SPECIFICATION

```
sig State { near, far: set Object }

action crossRiver[from, to  : set Object]
pre  { Farmer in from }
post { one x: from | {
        from' = from – x – Farmer – from’.eats &&
        to' = to + x + Farmer }
     }

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States
```

RIVER CROSS - DYNALLOY SPECIFICATION

```
sig State { near, far: set Object }


action crossRiver[from, to  : set Object]
pre  { Farmer in from }
post { one x: from | {
        from' = from – x – Farmer – from’.eats &&
        to' = to + x + Farmer }
     }

program solvePuzzle[near, far: set Object] {
    assume (Object in near && no far);
    (crossRiver[near, far] + crossRiver[far, near])*;
    [Object in far]?
}

run { last.far = Object } for 8 States
```

RIVER CROSS - DYNALLOY SPECIFICATION

```
sig State { near, far: set Object }

action crossRiver[from, to  : set Object]
pre  { Farmer in from }
post { one x: from | {
         from' = from – x – Farmer – from’.eats &&
         to' = to + x + Farmer }
     }

program solvePuzzle[near, far: set Object] {
   assume (Object in near && no far);
   (crossRiver[near, far] + crossRiver[far, near])*;
   [Object in far]?
}

run solvePuzzle for 4 lurs 8
```

RIVER CROSS - DYNALLOY
PARTIAL CORRECTNESS ASSERTIONS

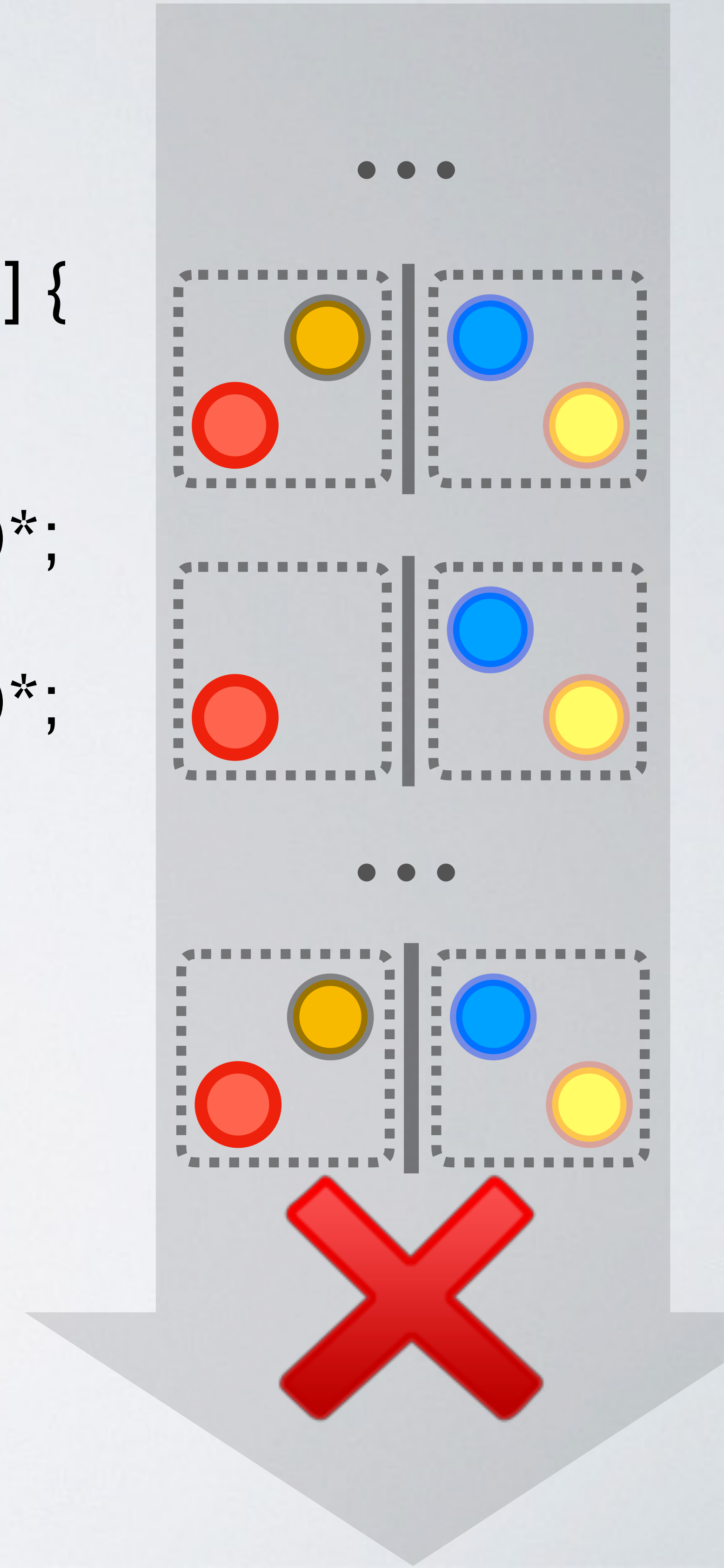
{ *precondition* }

PROGRAM

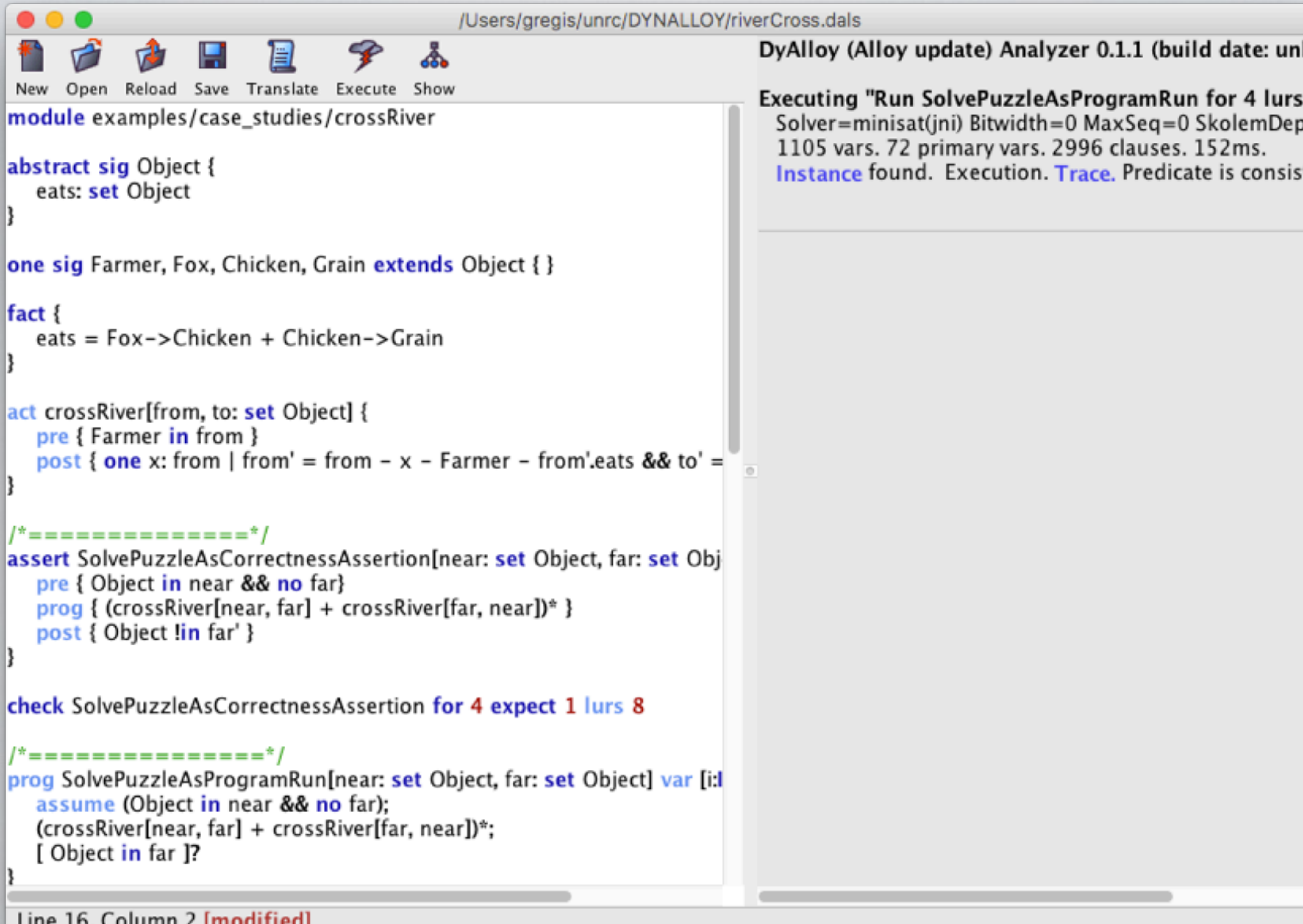
{ *postcondition* }

RIVER CROSS - DYNALLOY
PARTIAL CORRECTNESS ASSERTIONS

```
assert noResurrection[near, far: set Object, x: Object] {
    pre { no (near & far) }
    prog {
            (crossRiver[near, far] + crossRiver[far, near])*;
            [x !in (near+far)] ? ;
            (crossRiver[near, far] + crossRiver[far, near])*;
    }
    post { x !in (near'+far') }
}

check noResurrection for 4 lurs 8
```



DYNALLOY FEATURES



DYNALLOY FEATURES

- **Completely integrated into Alloy Analyzer**
- **Fully compatible with standard Alloy, produces detailed compile-time error reports**
- **Supports abstract syntax of programs, as well as imperative programming constructs (assignment, while loops, subprogram calls, …)**
- **Trace visualization (in the style of a program debugger)**

DYNALLOY FEATURES

- **Completely integrated into Alloy Analyzer**
- **Fully compatible with standard Alloy, produces detailed compile-time error reports**
- **Supports abstract syntax of programs, as well as imperative programming constructs (assignment, while loops, subprogram calls, …)**
- **Trace visualization (in the style of a program debugger)**
- ***Next release:***
  - *Efficient characterization of traces using skolemization*
  - *Efficient real and integer arithmetical representation*
  - *Control flow graph visualization for analyzing execution traces*