Mastering Alloy

Alcino Cunha

Overloading

Overloading

- Fields in disjoint signatures can be *overloaded* (have the same name)
- *Ambiguity* errors may occur

Overloading example

```
abstract sig Object {}
sig Dir extends Object {
   contains : set Entry
}
sig File extends Object {}
one sig Root extends Dir {}
sig Entry {
   contains : one Object,
   has : one Name
}
sig Name {}
```

Constraints

```
fact {
   // All directories are referred to in at most one entry
   all d : Dir | lone contains.d

    // The root is not referred in any entry
   no contains.Root

   // All objects except the root are referred to in at least one entry
   Object - Root in Entry.contains

   // Different entries in a directory must have different names
   all d : Dir, n : Name | lone (d.contains & has.n)
}
```

Constraints

```
fact {
  // A directory cannot be contained in itself
  all d : Dir | d not in d.^(contains.contains)
}
```

Ambiguity errors

```
run { some contains }
```

```
A type error has occurred:
This name is ambiguous due to multiple matches:
field this/Dir <: contains
field this/Entry <: contains
```

Resolving ambiguities

```
run { some d : Dir | some d.contains }

run { some contains & Dir->Entry }

run { some Dir <: contains }
```

Predicates and functions

Predicates

- In Alloy *predicates* are parametrized reusable constraints
  - Can also be derived propositions (without arguments)
  - Parameters can be arbitrary relations
- Only hold when invoked in a fact, command, or other predicates
- Recursive definitions are not allowed
- Run commands can directly ask for an instance satisfying a predicate
  - Atoms instantiating the parameters are shown in the visualizer

Predicate example

```
pred isreachable [d : Dir, o : Object] {
  o in d.^(contains.refersTo)
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | not isreachable[d,d]
}
```

Higher-order predicate example

```
pred acyclic [r : univ -> univ] {
  no ^r & iden
}

fact {
  // A directory cannot be contained in itself
  acyclic[contains.refersTo]
}
```

Functions

- *Functions* are parametrized reusable expressions
  - Parameters can be arbitrary relations
- Functions without parameters can be used to define derived relations
  - These show up in the visualizer
- Recursive definitions are not allowed

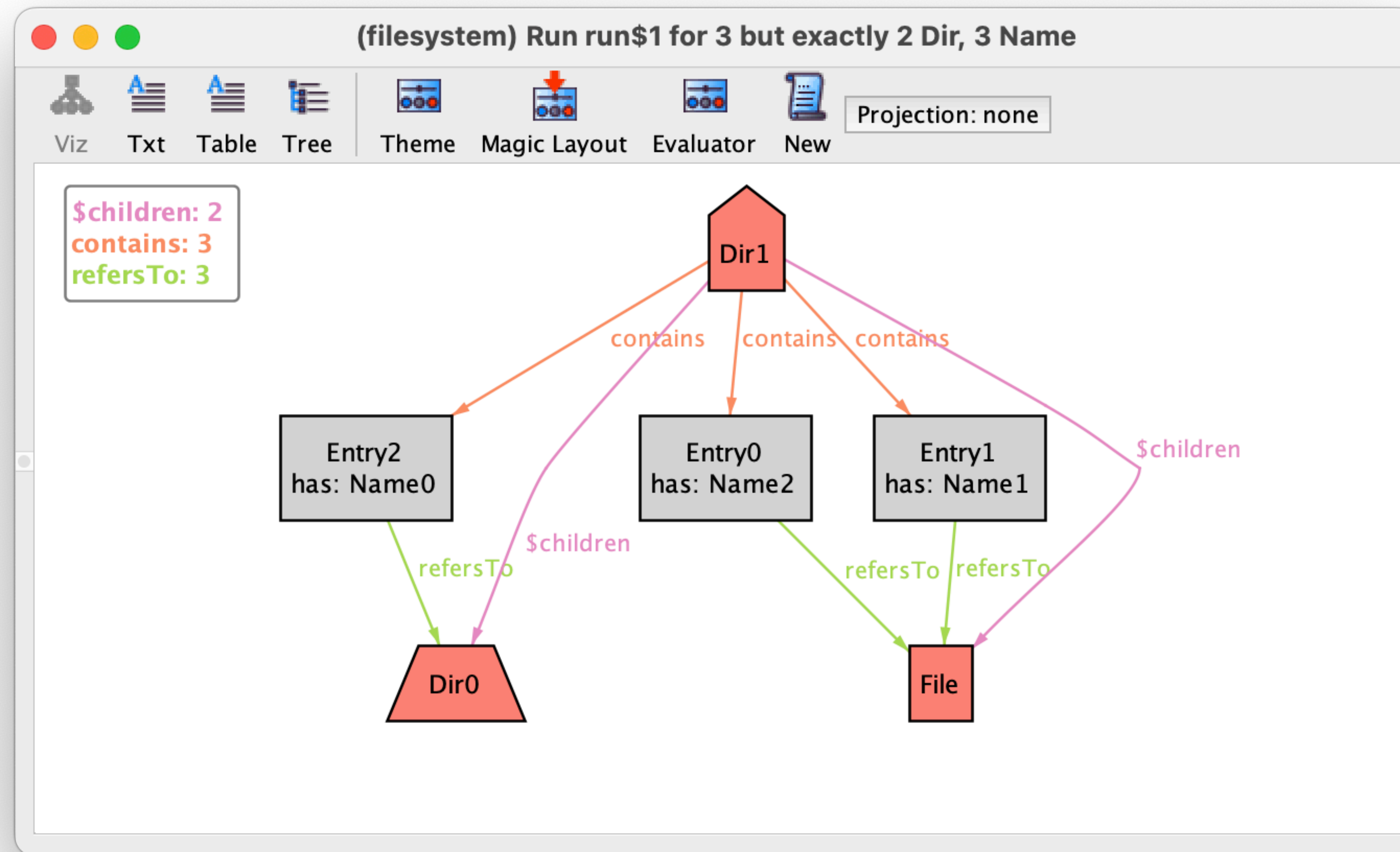
Function example

```
fun descendants [d : Dir] : set Object {
  d.^(contains.refersTo)
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | d not in descendants[d]
}
```

Derived relation example

```
fun children : Dir -> Object {
  contains.refersTo
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | d not in d.^children
}
```

Visualizing derived relations



Modules

Modules

- A specification can be split into *modules*
- The module name is declared in the first line with keyword **`module`**
- A module can be imported with an **`open`** statement
  - The imported model name must match the path of the corresponding file
  - To disambiguate a call to an entity, the module name can be prepended
  - An alias can be given with the **`as`** keyword
- A module can be parametrised by one or more signatures

Module example

```
module relation

pred acyclic [r : univ -> univ] {
  no ^r & iden
}
```

Module example

```
open relation

fact {
  // A directory cannot be contained in itself
  acyclic[contains.refersTo]
}
```

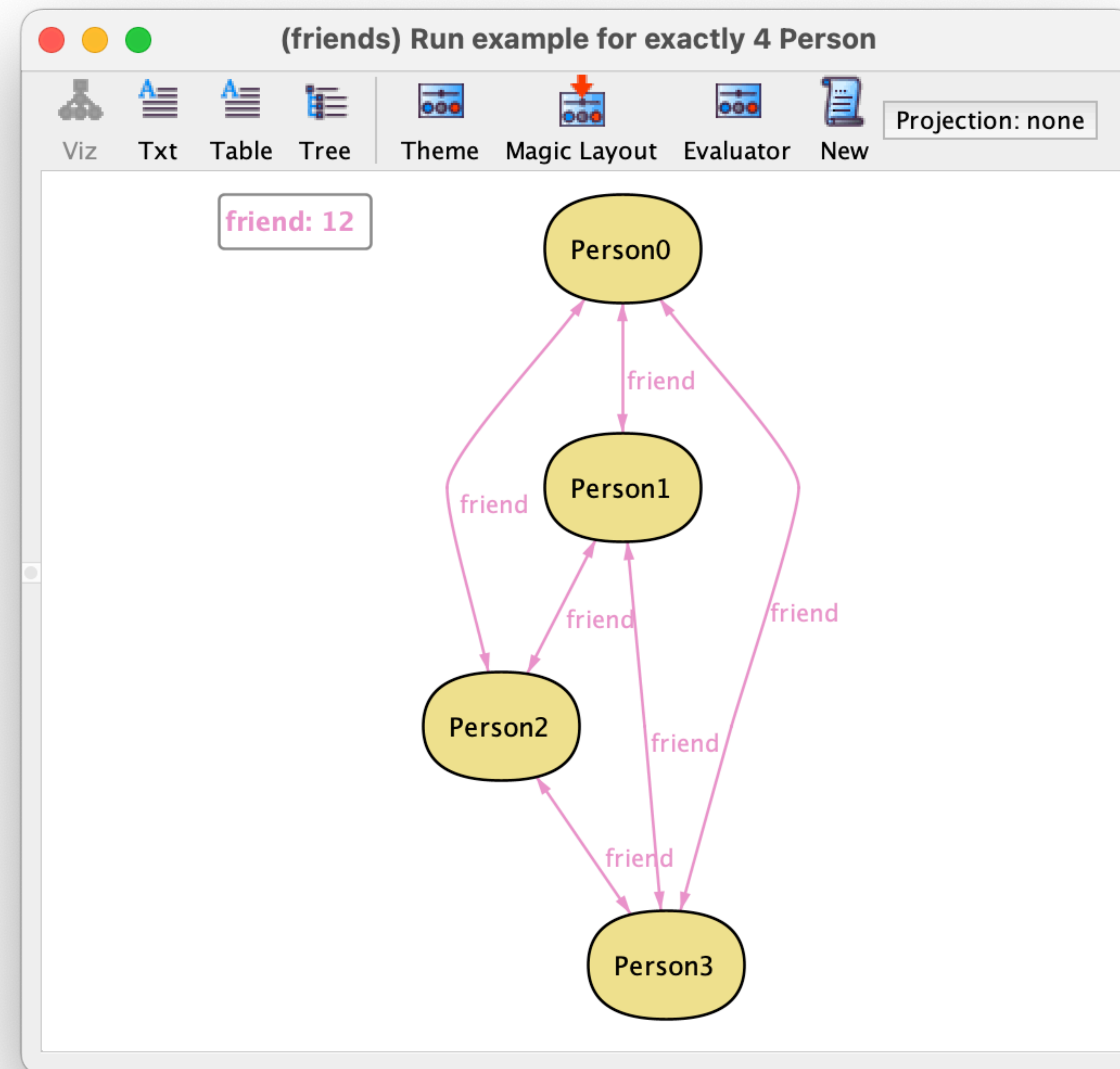
Parametrized module example

```
module graph[node]

pred complete[adj : node -> node] {
  all n : node | n.adj = node-n
}
```

Parametrized module example

```
open graph[Person]

sig Person {
  friend : set Person
}

fact { complete[friend] }
```

We are all friends



Predefined modules

`util/relation`	Useful functions and predicates for binary relations
`util/ternary`	Useful functions and predicates for ternary relations
`util/graph[A]`	Useful functions and predicates for graphs with nodes from signature `A`
`util/natural`	Natural numbers, including some arithmetic operations
`util/boolean`	Boolean type, including common logical connectives
`util/ordering[A]`	Imposes a total order on signature `A`

`util/ordering`

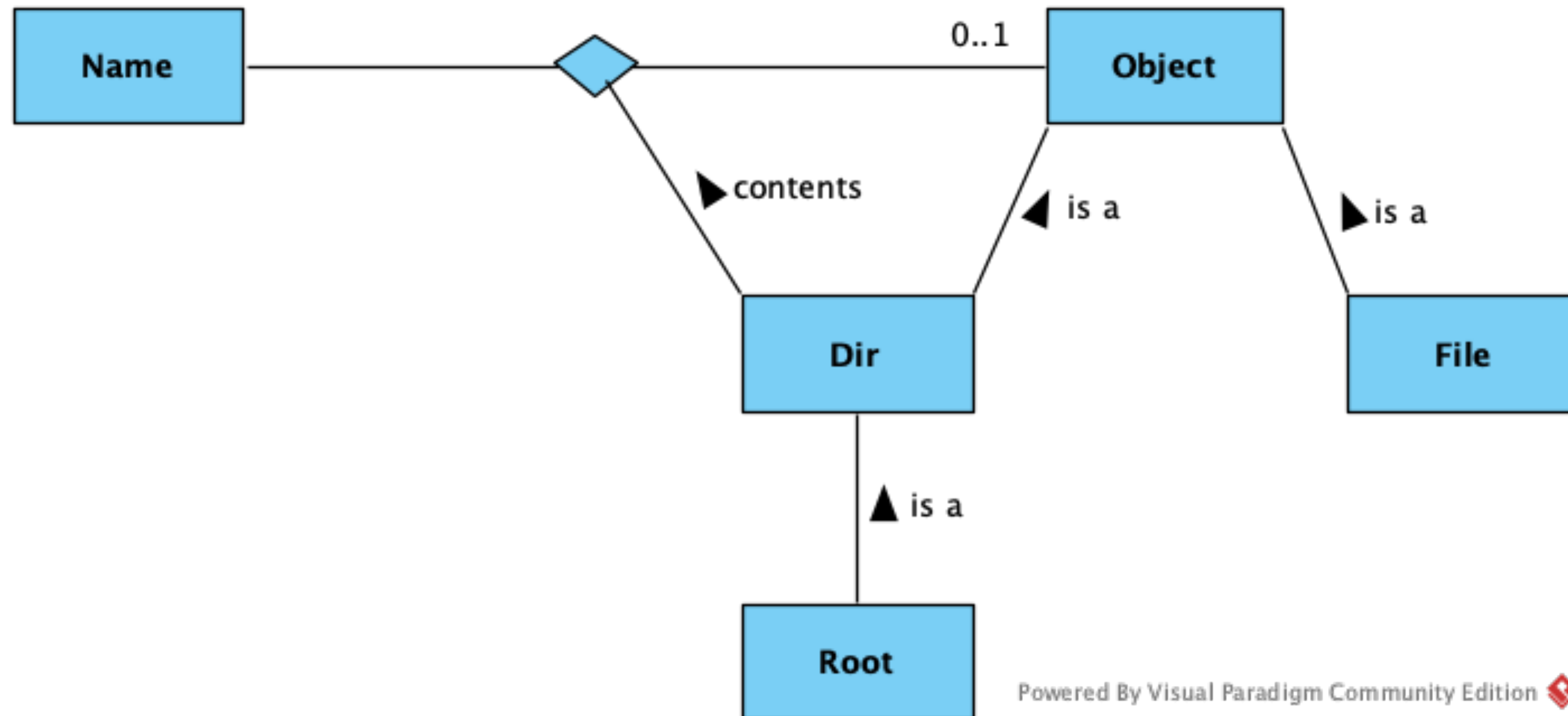
- Imposes a total order on the parameter signature
- For efficiency reasons the scope on that signature becomes exact
- Visualiser attempts to name atoms according to the order
- Many useful functions and relations, including
  - `next` and `prev` binary relations
  - `first` and `last` singleton sets
  - `lt`, `lte`, `gt`, and `gte` comparison predicates

util/ordering example

```
open util/ordering[Date]

sig Date {}
sig Entry {
   refersTo : one Object,
   has   : one Name,
   date : one Date
}

fact {
   // A directory cannot be contained in itself
   all e : Entry, c : e.refersTo.contains | lt[e.date, c.date]
}
```

N-ary relations

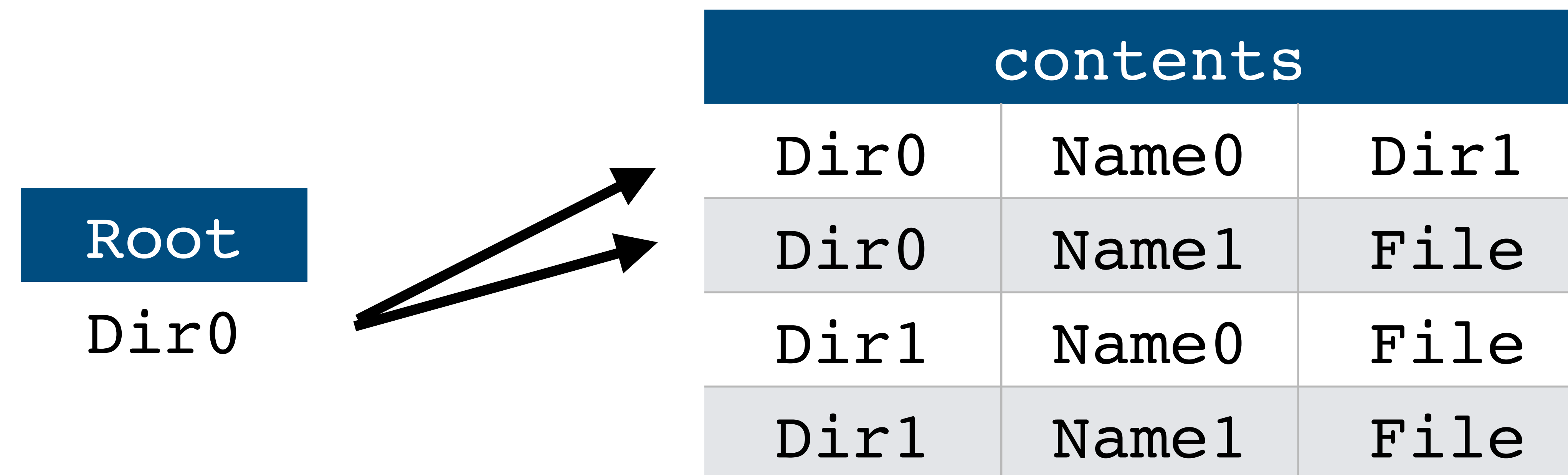
N-ary relationships *a la* UML



Ternary relation example

```
abstract sig Object {}
sig Dir extends Object {
   contents : Name -> lone Object
}
sig File extends Object {}
one sig Root extends Dir {}
sig Name {}
```

Composition



Root
Dir0

contents		
Dir0	Name0	Dir1
Dir0	Name1	File
Dir1	Name0	File
Dir1	Name1	File

Root . contents	
Name0	Dir1
Name1	File

Constraints

```
fact {
   // All directories are referred to in at most one entry
   all d : Dir | lone contents.d

   // The root is not referred in any entry
   no contents.Root

   // All objects except the root are referred to in at least one entry
   Object - Root in Name.(Dir.contents)
}
```

Constraints

```
fun children : Dir -> Object {
  ???
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | d not in d.^children
}
```

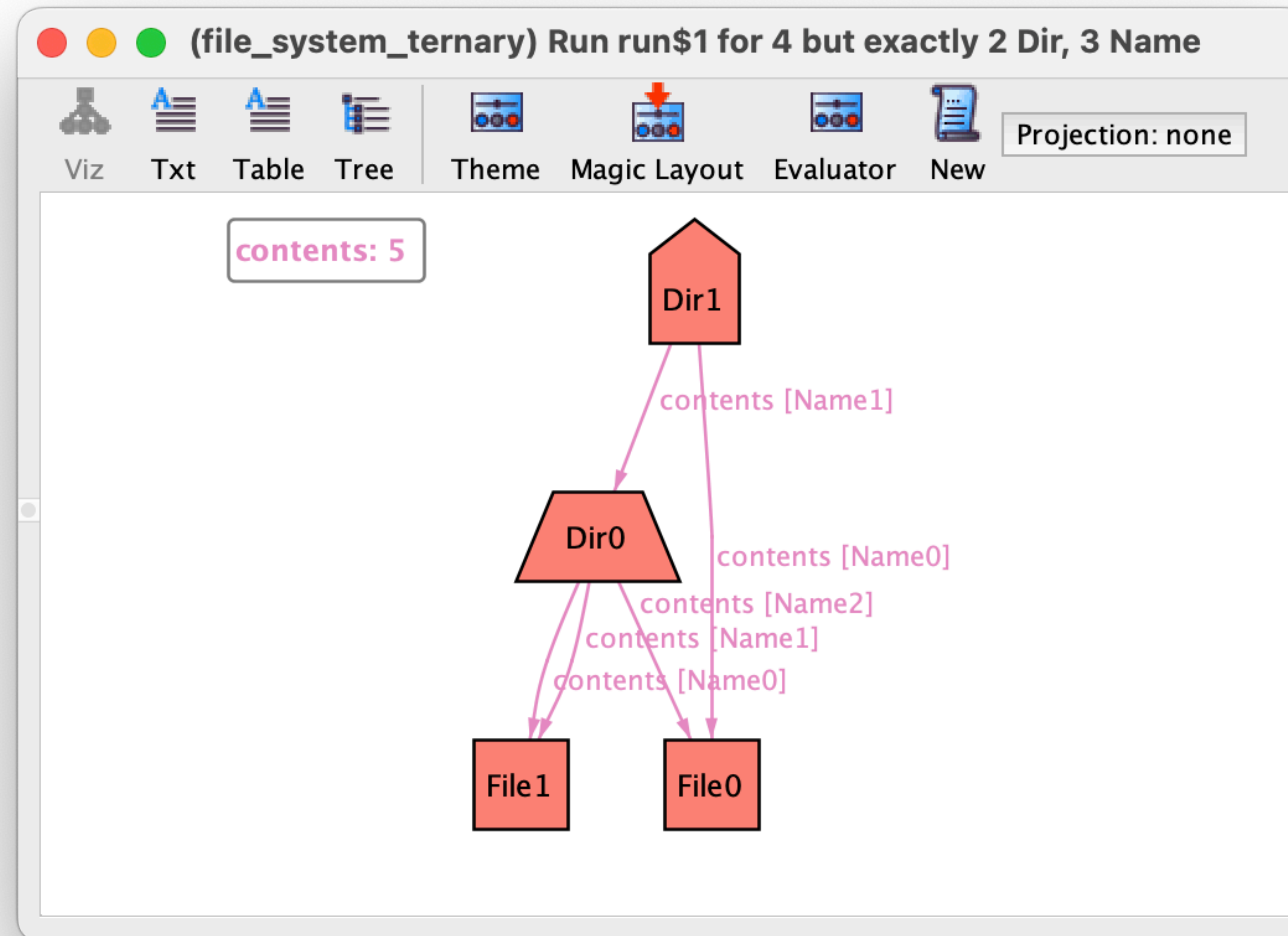
Comprehension

$$\{\ x_1 : A_1, \ldots, x_n : A_n \ \mid\ \phi\ \}$$

Constraints

```
fun children : Dir -> Object {
  { x : Dir, y : Object | some x.contents.y }
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | d not in d.^children
}
```

Visualizing N-ary relations



Type system

Arity error

```
run { some has & File }
```

```
A type error has occurred:
& can be used only between 2 expressions of the same arity.
Left type = {this/Entry->this/Name}
Right type = {this/File}
```

Ambiguity error

```
run { some contains }
```

```
A type error has occurred:
This name is ambiguous due to multiple matches:
field this/Dir <: contains
field this/Entry <: contains
```

Irrelevance warning

```
run { some Dir.has }
```

```
Warning #1
The join operation here always yields an empty set.
Left type = {this/Dir}
Right type = {this/Entry->this/Name}
```

Test automation

NetworkX

networkx.org

NetworkX
Network Analysis in Python

Install Tutorial Reference Gallery Developer Releases Guides

Search ⌘ + K

3.3 (stable)

Simple Paths
Small-world
s metric
Sparsifiers
Structural holes
Summarization
Swap
Threshold Graphs
Time dependent
Tournament
Traversal
Tree
Triads
Vitality
Voronoi cells
Walks
Wiener Index
Functions
Graph generators
Linear algebra
Converting to and from other data formats
Relabeling nodes
Reading and writing graphs
Drawing
Randomness
Exceptions

`is_arborescence(G)` [source]

Returns True if `G` is an arborescence.

An arborescence is a directed tree with maximum in-degree equal to 1.

Parameters:

G : *graph*

The graph to test.

Returns:

b : *bool*

A boolean that is True if `G` is an arborescence.

See also

`is_tree`

Notes

In another convention, an arborescence is known as a *tree*.

Examples

```
>>> G = nx.DiGraph([(0, 1), (0, 2), (2, 3), (3, 4)])
>>> nx.is_arborescence(G)
True
>>> G.remove_edge(0, 1)
>>> G.add_edge(1, 2)  # maximum in-degree is 2
>>> nx.is_arborescence(G)
False
```


On this page
`is_arborescence()`

NetworkX

networkx.org

NetworkX
Network Analysis in Python

Install Tutorial Reference Gallery Developer Releases Guides

Search ⌘ + K

3.3 (stable)

Coloring
Communicability
Communities
Components
Connectivity
Cores
Covering
Cycles
Cuts
D-Separation
Directed Acyclic Graphs
Distance Measures
Distance-Regular Graphs
Dominance
Dominating Sets
Efficiency
Eulerian
Flows
Graph Hashing
Graphical degree sequence
Hierarchy
Hybrid
Isolates
Isomorphism
Link Analysis
Link Prediction

`strongly_connected_components(G)` [source]

Generate nodes in strongly connected components of graph.

Parameters:

G : ***NetworkX Graph***

A directed graph.

Returns:

comp : ***generator of sets***

A generator of sets of nodes, one for each strongly connected component of G.

Raises:

NetworkXNotImplemented

If G is undirected.

See also

`connected_components`
`weakly_connected_components`
`kosaraju_strongly_connected_components`

Notes

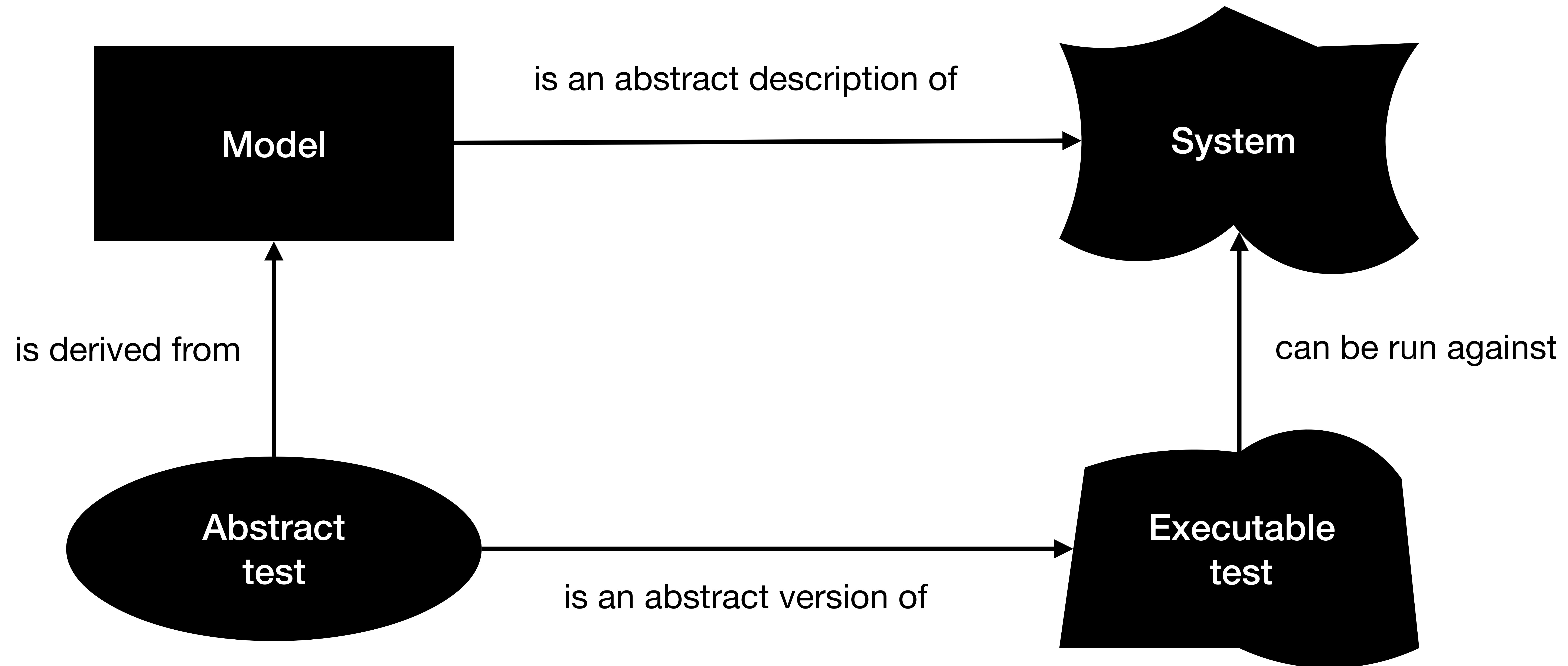
Uses Tarjan's algorithm[R827335e01166-1]_ with Nuutila's modifications[R827335e01166-2]_. Nonrecursive version of algorithm.

References

[1] Depth-first search and linear graph algorithms, R. Tarjan SIAM Journal of Computing

On this page
`strongly_connected_components()`

Model-based testing



Arborescence

```
some sig Node {
    adj : set Node
}

pred isArborescence {
    // In-degree at most one
    all n : Node | lone adj.n
    // No cycles
    all n : Node | n not in n.^adj
    // All nodes can be reached from a root
    some n : Node | Node-n in n.^adj
}

run {isArborescence} for 7
run {not isArborescence} for 7
```

Components

```
sig Node {
   adj : set Node,
   component : one Component
}
sig Component {}

fact {
  // Nodes belong to same component iff they are reachable from each other
  all x,y : Node | x.component = y.component iff x in y.*adj and y in x.*adj
  // Only return necessary components
  Component = Node.component
}

run {} for 6
```

Let’s do it with Alloy API!