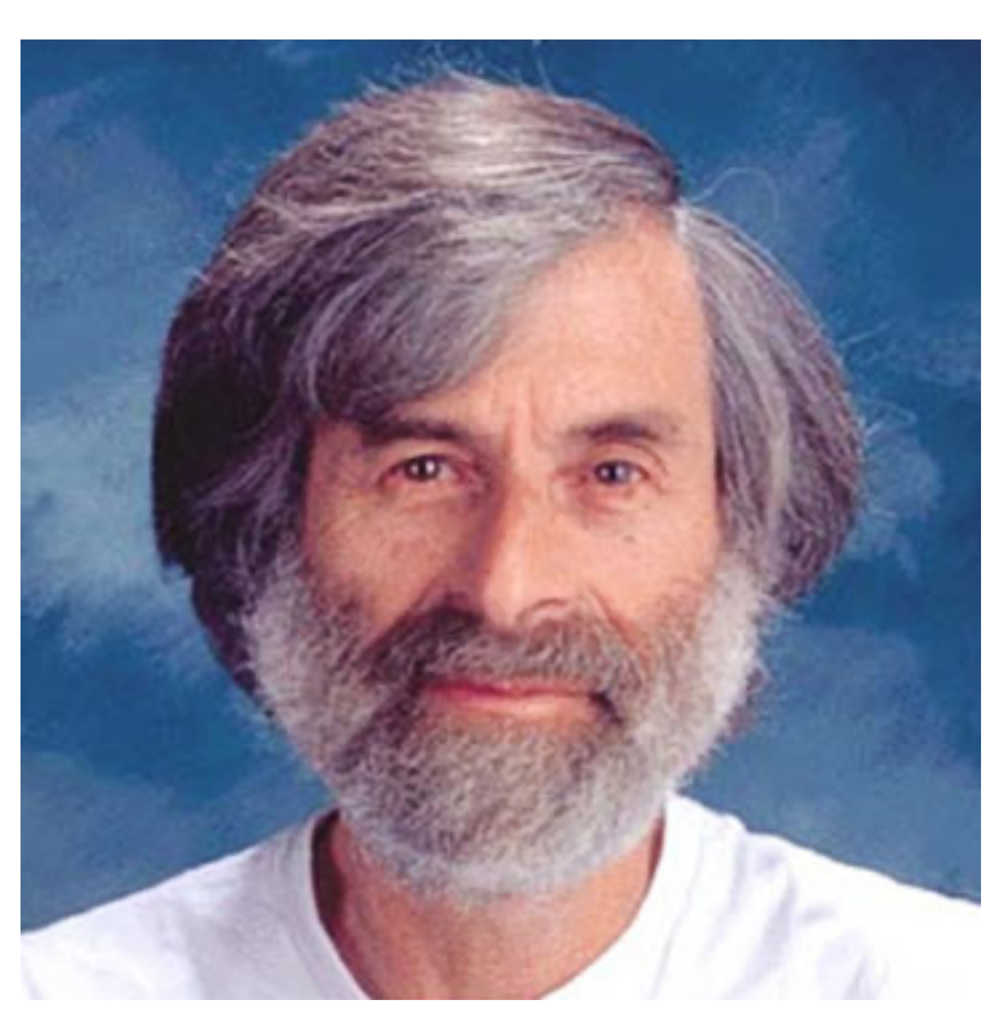
Structural design with Alloy

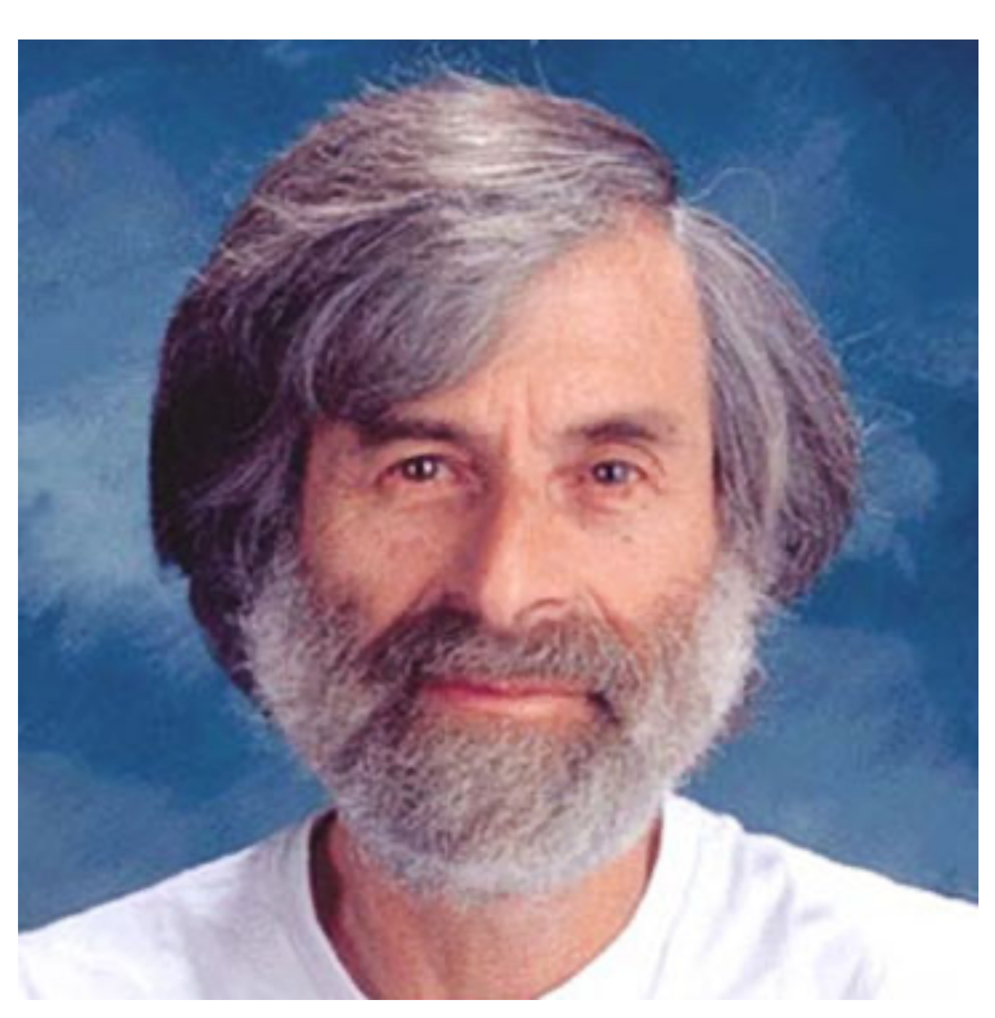
Alcino Cunha

"A *specification* is a written description of what a system is supposed to do. Specifying a system helps us understand it. It's a good idea to understand a system before building it, so it's a good idea to write a specification of a system before implementing it."



–Leslie Lamport

“A specification is an *abstraction*. It describes some aspects of the system and ignores others.”



–Leslie Lamport

“The core of software development [...] is the *design* of abstractions. An abstraction is [...] an idea reduced to its essential form.”

–Daniel Jackson

lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

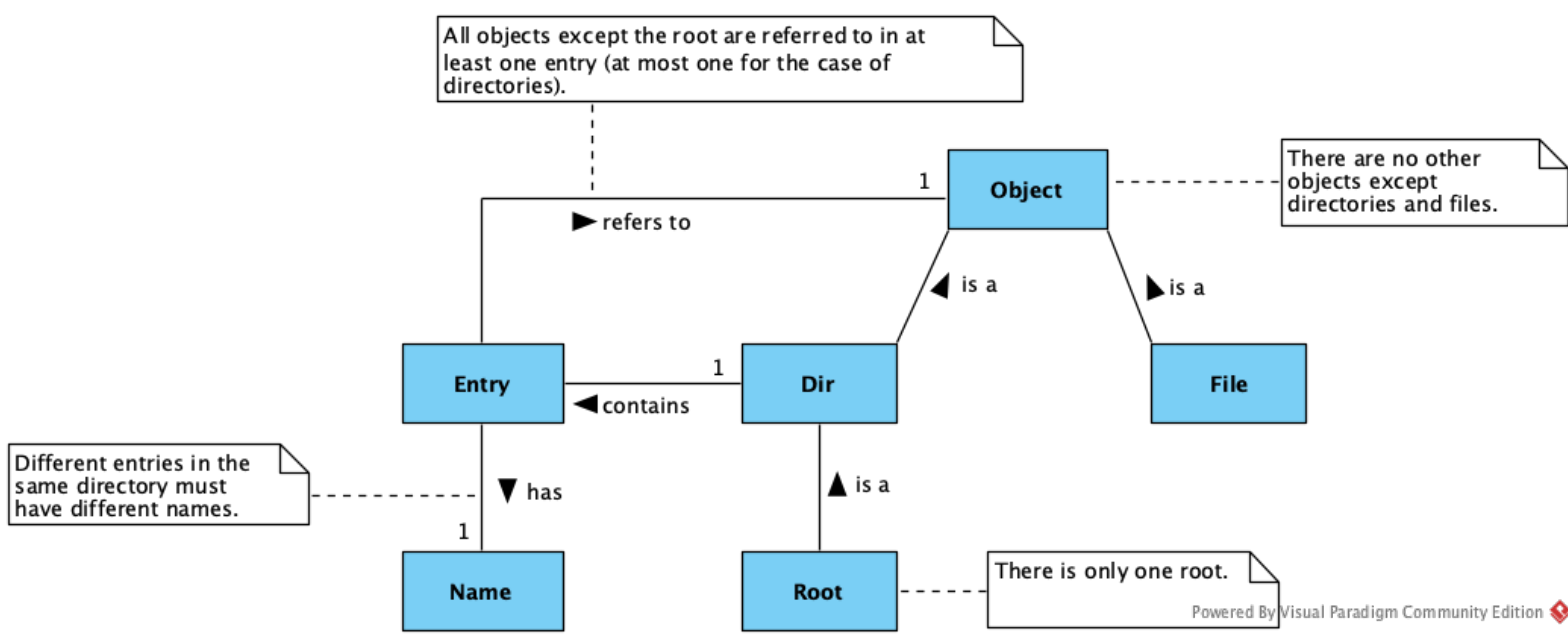
Software structures

- Data structures
- Database schemas
- Architectures
- Network topologies
- Ontologies
- Domain models

Structural design

- Understand entities and their relationships
- Elicit requirements
- Explore alternatives

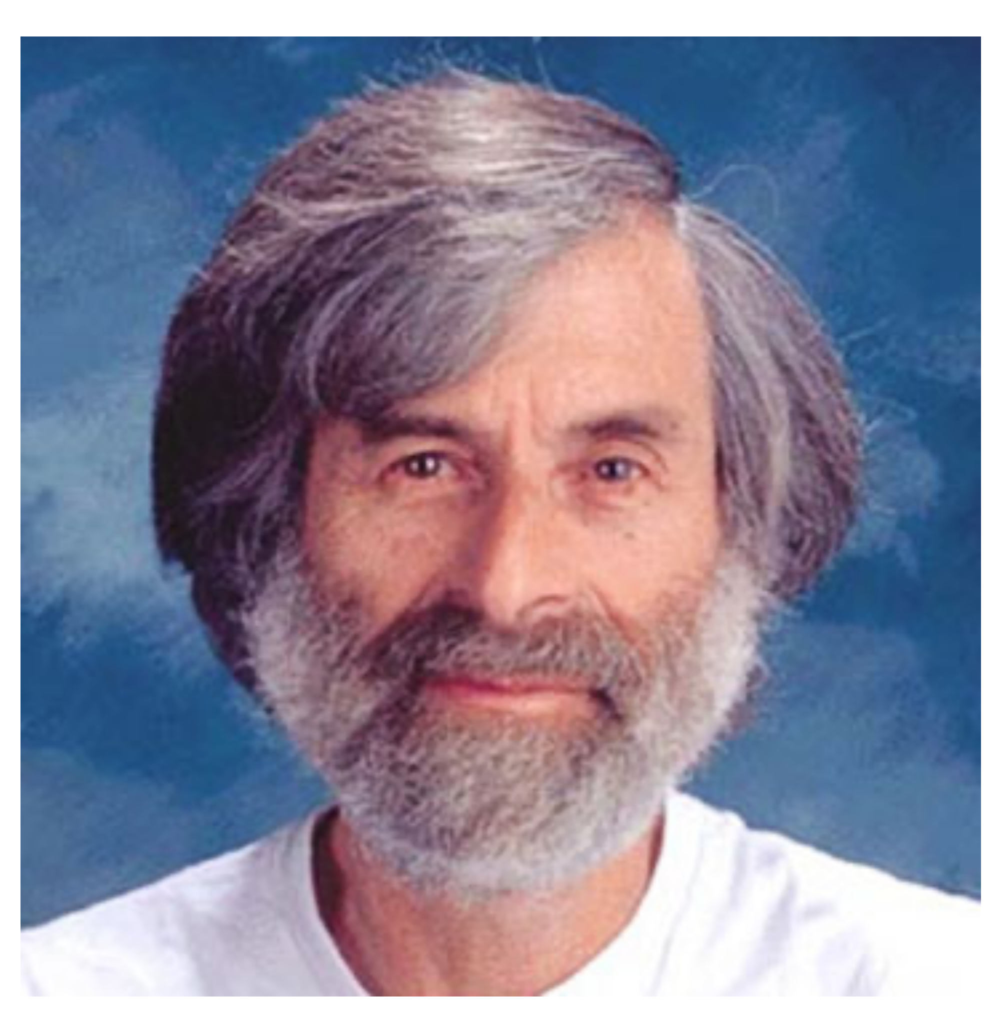
Domain modeling *a la* UML



Domain modeling *a la* UML

- How to validate the model?
- Any forgotten or redundant constraints?
- What exactly mean the constraints?
- Do the constraints entail all the expected properties?

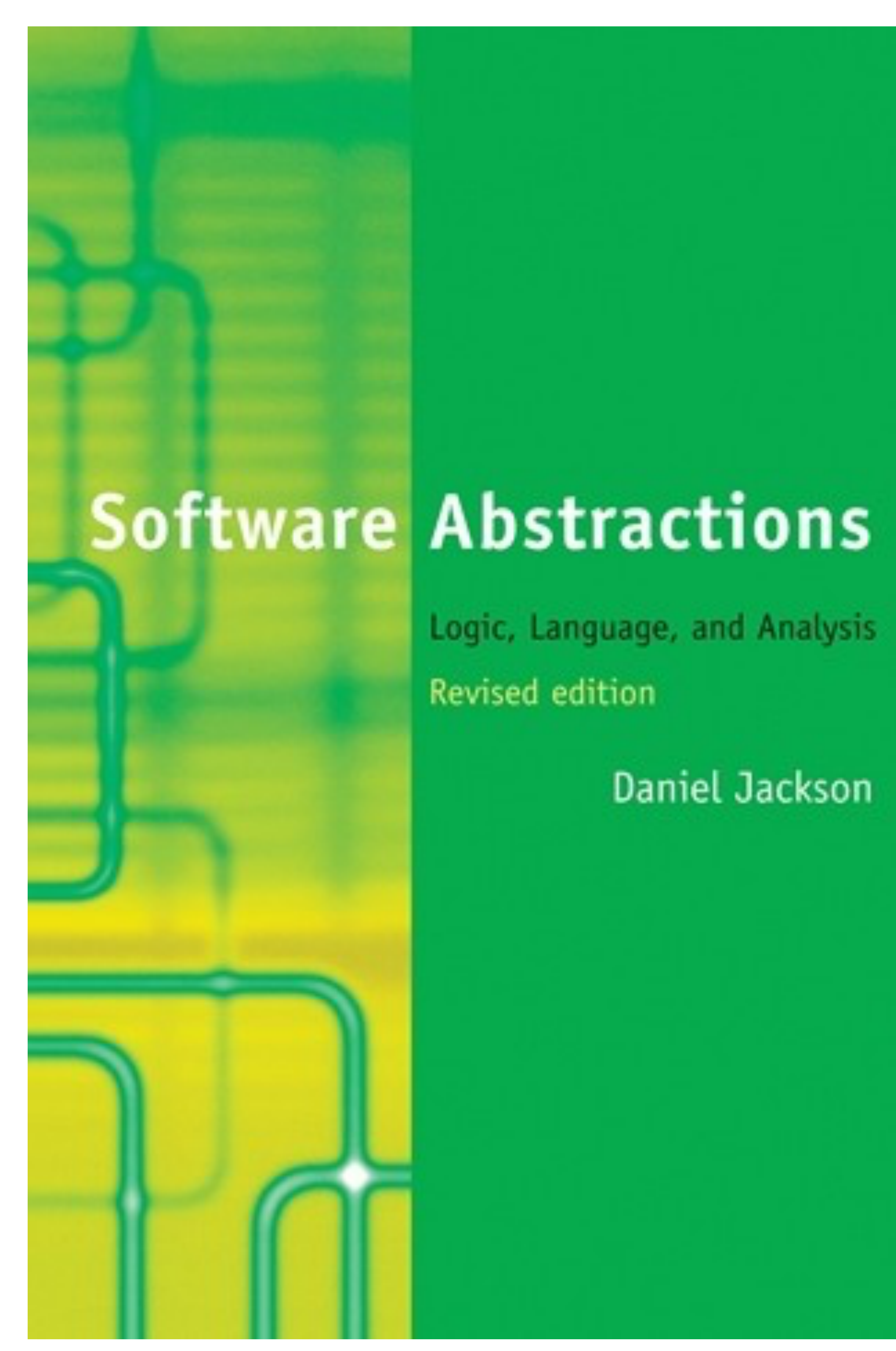
“Our basic tool for writing specifications is *mathematics*. Mathematics is nature’s way of letting you know how slopping your writing is. It’s hard to be precise in an imprecise language like English or Chinese. In engineering, imprecision can lead to errors. To avoid errors, science and engineering have adopted mathematics as their language.”



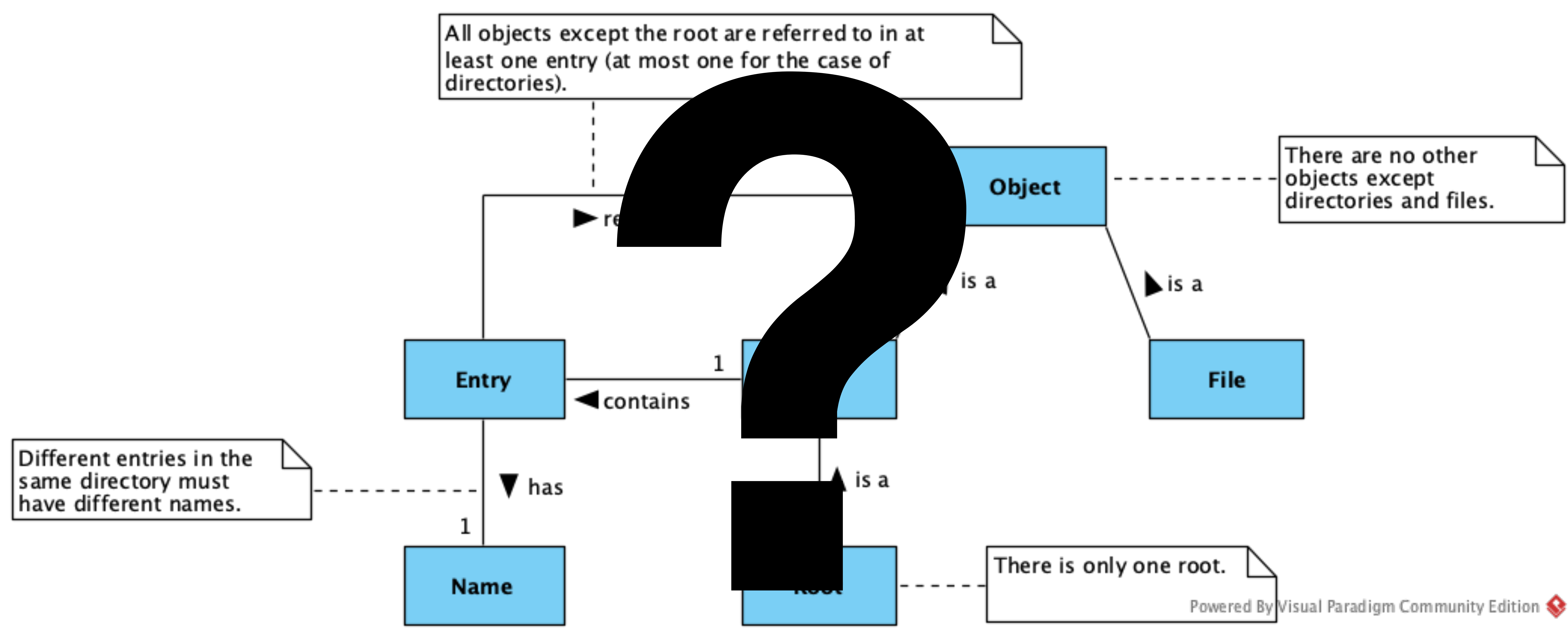
–Leslie Lamport

Software design with Alloy

- Alloy is a formal modeling language
- Can be used to declare structures and specify constraints
- Models can be automatically analyzed
- Tailored for abstraction


Software Abstractions
Logic, Language, and Analysis
Revised edition
Daniel Jackson

Domain modeling with Alloy



First order logic

Types ➟ Signatures

```
sig Object {}
sig Entry {}
sig Name {}
```

Unary predicates ➟ Subset signatures

```
sig Dir in Object {}
sig File in Object {}
```

Constants ➠ Unary predicates

```
sig Root in Object {}
```

Signatures

- *Signatures* are sets
- Inhabited by *atoms* from a finite *universe* of discourse
- *Top-level* signatures are disjoint

Binary predicates ➟ Fields

```
sig Object {
  contains : set Entry
}
sig Entry {
  refersTo : set Object,
  name : set Name
}
```

Fields

- *Fields* are relations
- Inhabited by sets of *tuples* of atoms from the universe
- Fields are subsets of the Cartesian product of the source and target type signatures
- All tuples in a field have the same *arity*

Facts

- *Facts* specify assumptions

```
fact { φ }
```

- Facts can be named

```
fact Name { φ }
```

- A single fact can have several constraints, one per line

```
fact {
  φ
  ψ
}
```

Subtyping constraints in FOL

```
fact {
  // Files and directories are disjoint
  ∀o : Object. File(o) → ¬Dir(o)
  // There are no other objects except directories and files
  ∀o : Object. File(o) ∨ Dir(o)
  // Root is a directory
  ∀o : Object. Root(o) → Dir(o)
  // Only directories contain entries
  ∀o : Object, e : Entry. contains(o, e) → Dir(o)
}
```

Multiplicity constraints in FOL

```
fact {
  // Each entry has one name
```
$\forall e : \texttt{Entry} \,.\, \exists n : \texttt{Name} \,.\, \texttt{name}(e, n)$

$\forall e : \texttt{Entry}, n_1, n_2 : \texttt{Name} \,.\, \texttt{name}(e, n_1) \land \texttt{name}(e, n_2) \rightarrow n_1 = n_2$

```
  // Each entry has one object
```
$\forall e : \texttt{Entry} \,.\, \exists o : \texttt{Object} \,.\, \texttt{refersTo}(e, o)$

$\forall e : \texttt{Entry}, o_1, o_2 : \texttt{Object} \,.\, \texttt{refersTo}(e, o_1) \land \texttt{refersTo}(e, o_2) \rightarrow o_1 = o_2$

```
  // Each entry is contained in one directory
```
$\forall e : \texttt{Entry} \,.\, \exists o : \texttt{Object} \,.\, \texttt{contains}(o, e)$

$\forall e : \texttt{Entry}, o_1, o_2 : \texttt{Object} \,.\, \texttt{contains}(o_1, e) \land \texttt{contains}(o_2, e) \rightarrow o_1 = o_2$

```
  // There is only one root
```
$\exists o : \texttt{Object} \,.\, \texttt{Root}(o)$

$\forall o_1, o_2 : \texttt{Object} \,.\, \texttt{Root}(o_1) \land \texttt{Root}(o_2) \rightarrow o_1 = o_2$

```
}
```

Subtypes ➟ Extension signatures

```
sig Dir extends Object {
   contains : set Entry
}
sig File extends Object {}
sig Root extends Dir {}
```

Abstract signatures

```
abstract sig Object {}
```

Signatures

- An *extension* signature is a subset of the *parent* signature
- Sibling extension signatures are disjoint
- We can declare fields in extension signatures
- All atoms in an *abstract* signature belong to one of its extensions
- The extensions partition the parent signature

Multiplicities

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

Multiplicities

- Multiplicity constraints can be directly imposed in signatures
- And on the target signature of fields
- Possible multiplicities are **`one`**, **`lone`**, **`some`**, and **`set`**
- The default in fields is **`one`** and in signatures is **`set`**
- Signatures with multiplicity **`one`** denote constants

Commands

- Alloy has two types of analysis *commands*:
  - `run { φ }` asks for an *example* that satisfies φ
  - `check { φ }` asks for a *counter-example* that refutes assertion φ
- Likewise facts, commands can be named and can have several constraints, one per line
- In the visualizer it is possible to ask for more examples or counter-examples by pressing *New*

Instances

- Both examples and counter-examples are instances of the model
- An *instance* is a valuation to all the signatures and fields
- An instance must satisfy the declarations and all the facts
- In an instance “everything is a relation”
    - Signatures are unary relations (sets of unary tuples)
    - Constants are singleton unary relations (sets with one unary tuple)

Scopes

- To ensure decidability commands have a *scope*
- The scope imposes a limit on the size of the (finite) universe the Analyzer will exhaustively explore
- The default scope imposes a limit of 3 atoms per top-level signature
- **`for`** can be used to specify a different scope for top-level signatures
- **`but`** can be used to specify different scopes for specific signatures
- **`exactly`** can be used to specify exact scopes

The small scope hypothesis

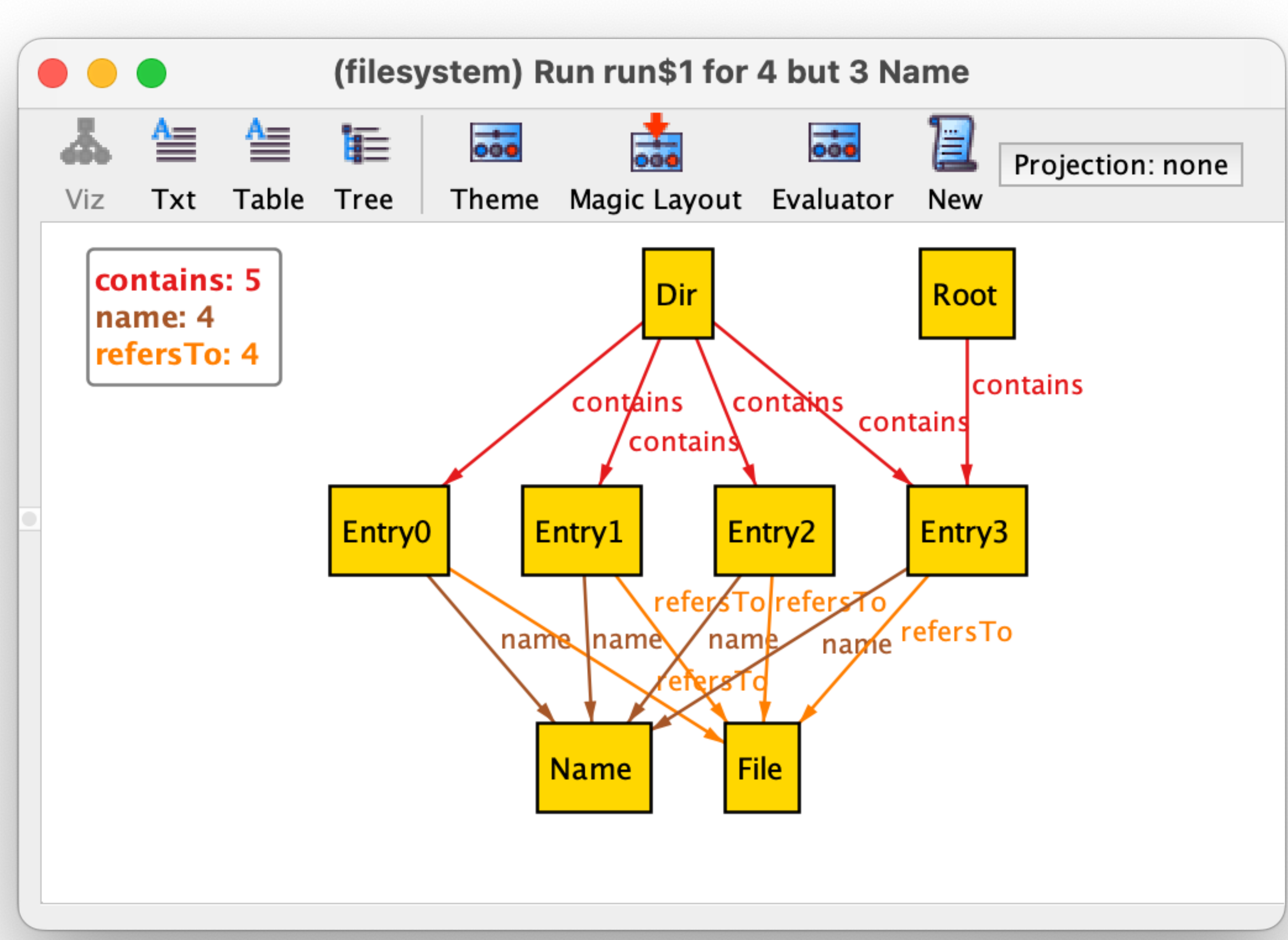
- If `run { φ }` returns an instance then φ is *consistent*, else φ **may be *inconsistent***
  - Could be consistent with a bigger scope!
- If `check { φ }` returns an instance then φ is *invalid,* else φ **may be *valid***
  - Could be invalid with a bigger scope!!!
- Anecdotical evidence suggests that most invalid assertions (or consistent predicates) can be refuted (or witnessed) with a small scope

A simple command

```
run {} for 4 but 3 Name
```

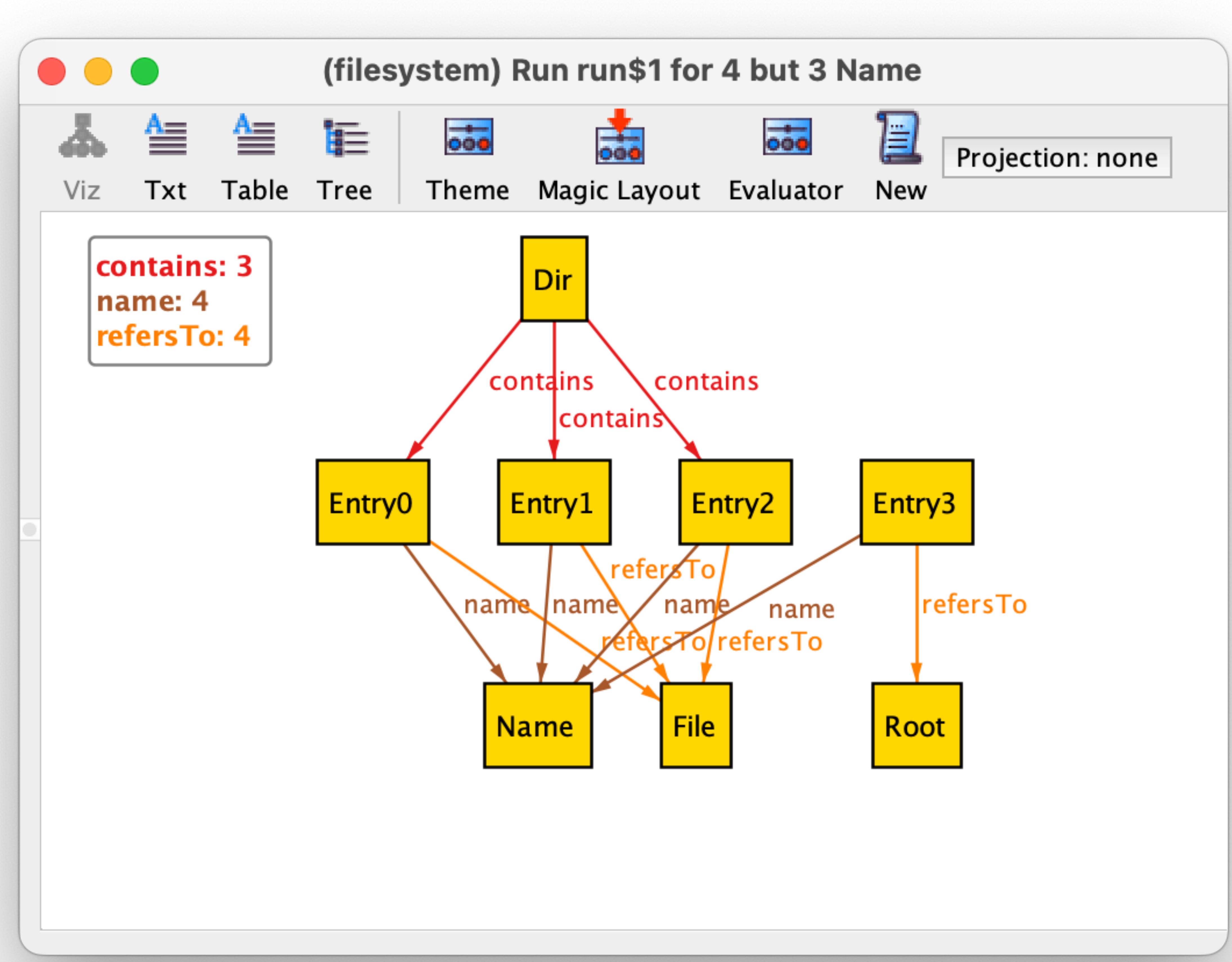
A simple command

```
run {} for 4 but 3 Name
```

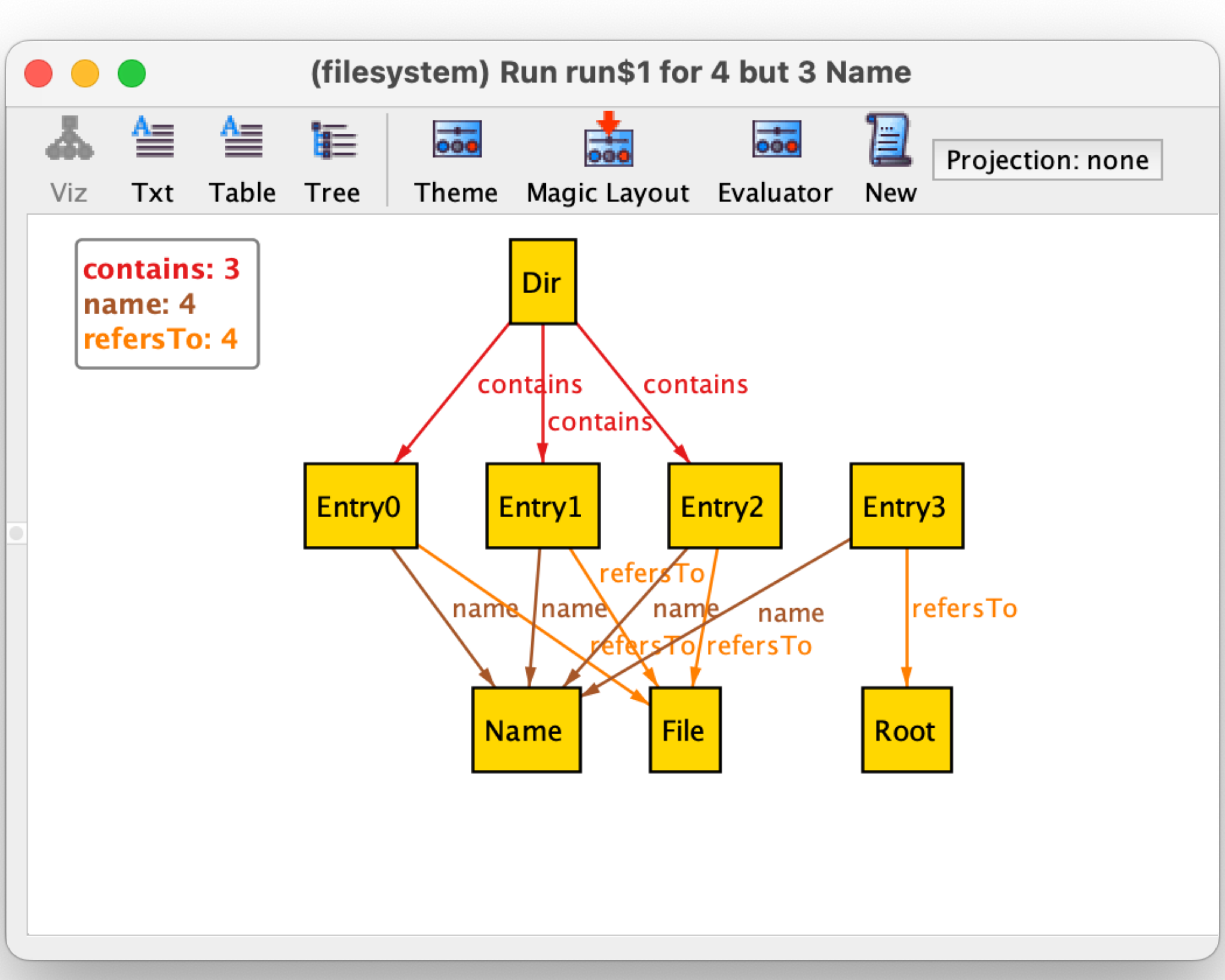


A simple command

```
run {} for 4 but 3 Name
```



Instances as graphs



Instances as relations

```
Object   = {(Dir),(File),(Root)}
Dir      = {(Dir),(Root)}
File     = {(File)}
Root     = {(Root)}
Entry    = {(Entry0),(Entry1),(Entry2),(Entry3)}
Name     = {(Name)}
contains = {(Dir,Entry0),(Dir,Entry1),(Dir,Entry2)}
refersTo = {(Entry0,File),(Entry1,File),(Entry2,File),(Entry3,Root)}
name     = {(Entry0,Name),(Entry1,Name),(Entry2,Name),(Entry3,Name)}
```

Instances as tables

Object
Dir
File
Root

Dir
Dir
Root

Root
Root

File
File

Name
Name

Entry
Entry0
Entry1
Entry2
Entry3

contains	
Dir	Entry0
Dir	Entry1
Dir	Entry2

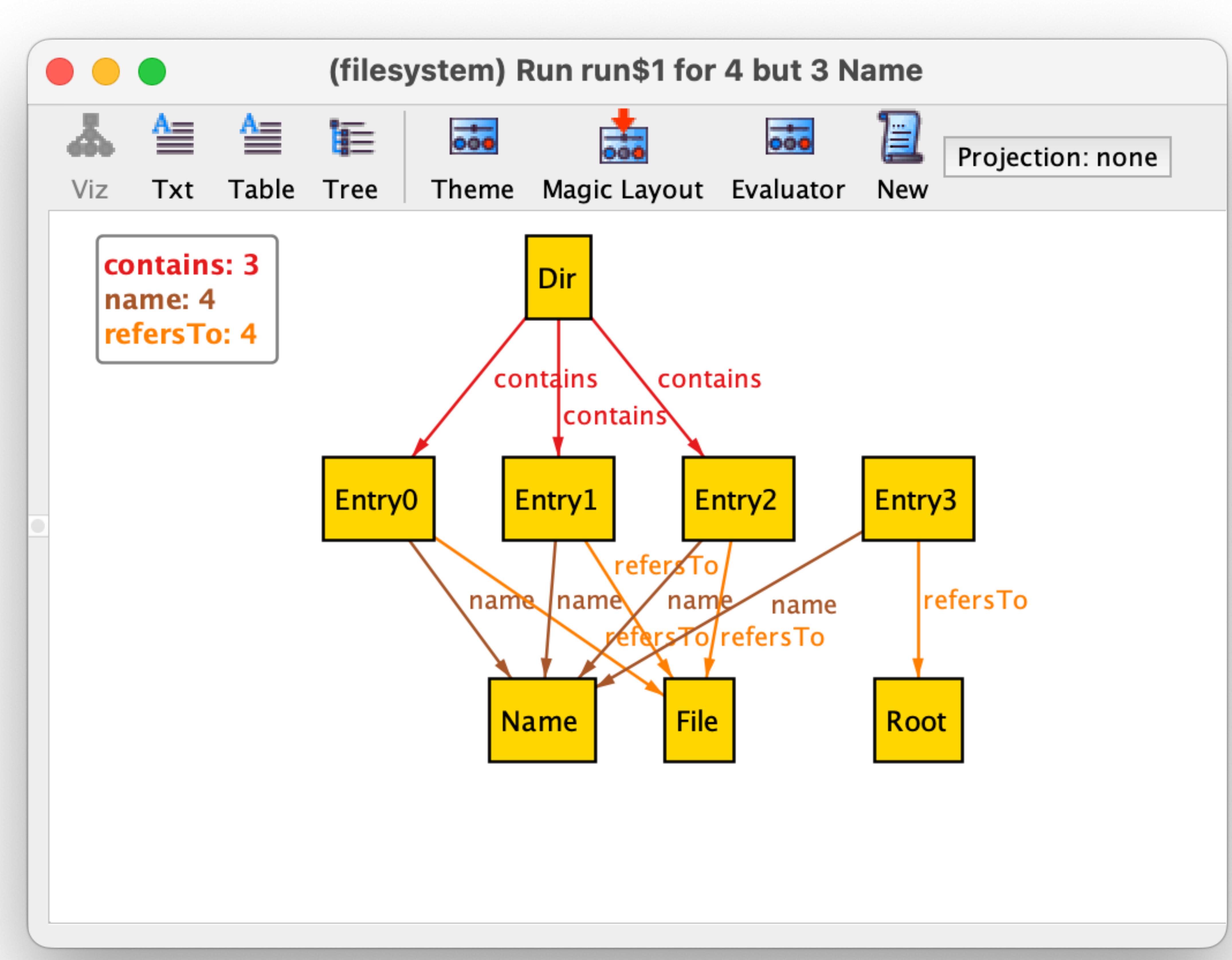
refersTo	
Entry0	File
Entry1	File
Entry2	File
Entry3	Root

name	
Entry0	Name
Entry1	Name
Entry2	Name
Entry3	Name

Atoms

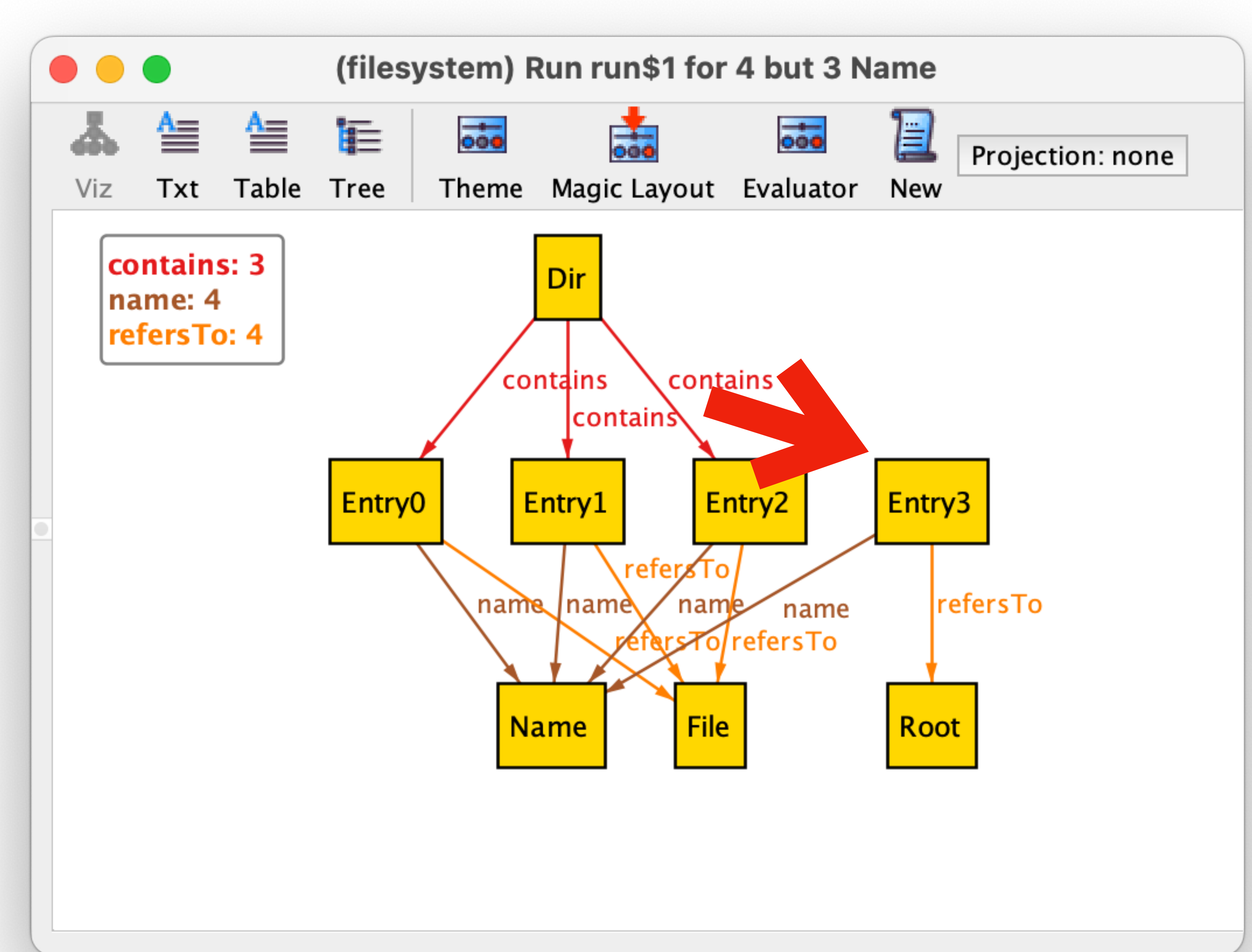
- The universe of discourse contains *atoms*
- Atoms are *uninterpreted* (no semantics)
- Named automatically according to the respective signatures
- Two instances are *isomorphic* (or *symmetric*) if they are equal modulo renaming
- The analysis implements a *symmetry breaking* mechanism to avoid returning isomorphic instances

Missing constraints



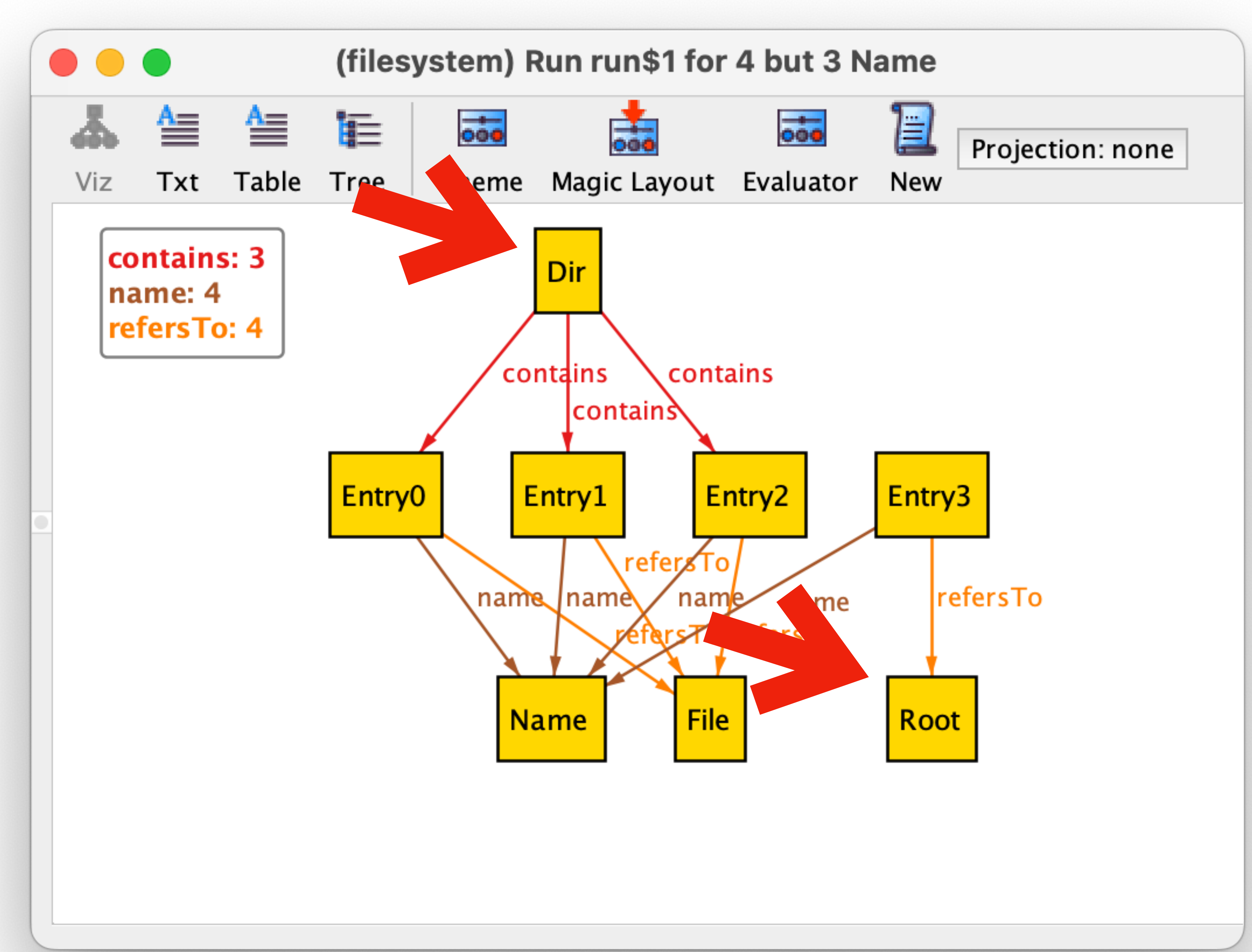
Missing constraints

- Each entry is contained in one directory



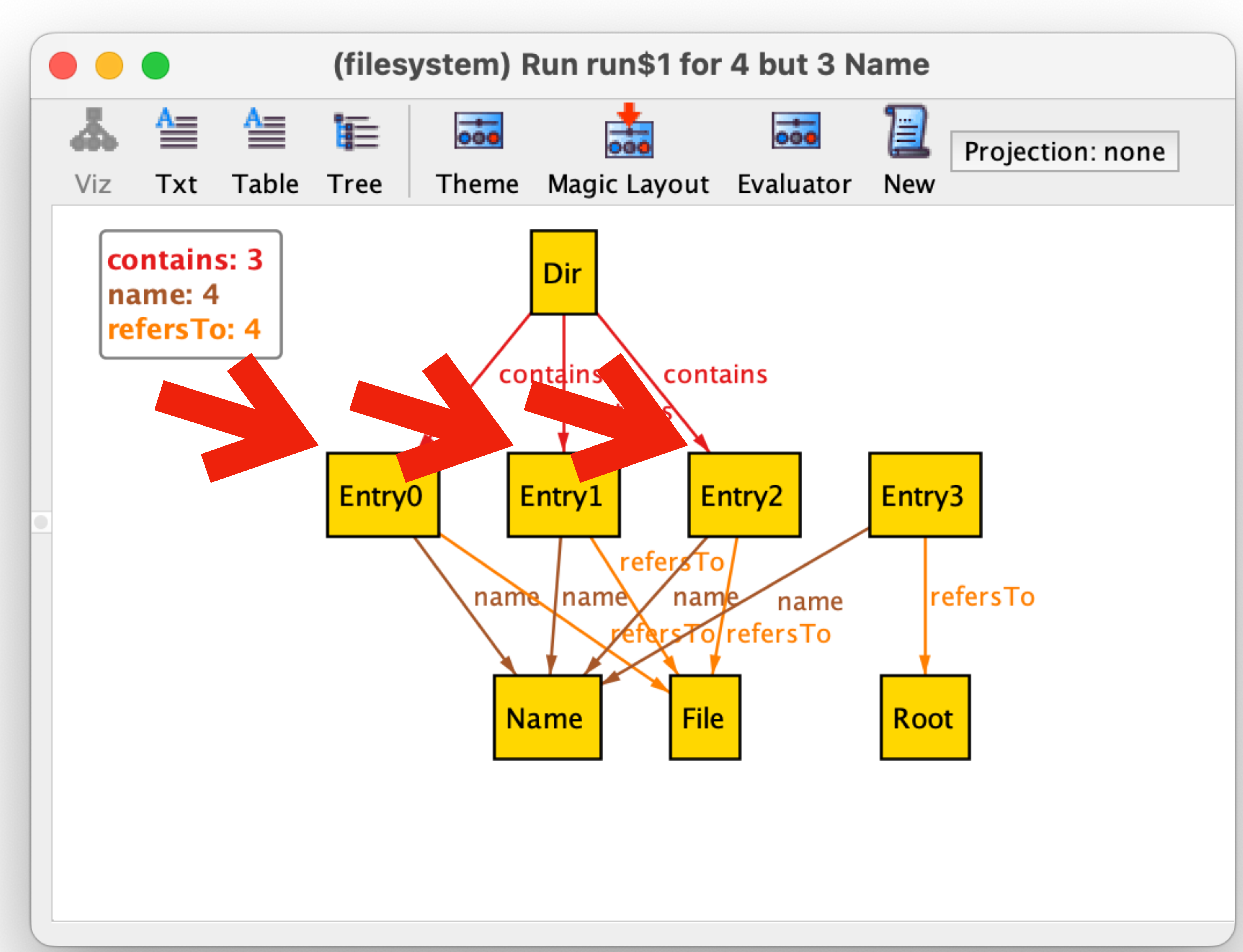
Missing constraints

- Each entry is contained in one directory
- All objects except the root are referred to in at least one entry (at most one for the case of directories)



Missing constraints

- Each entry is contained in one directory
- All objects except the root are referred to in at least one entry (at most one for the case of directories)
- Different entries in a directory must have different names

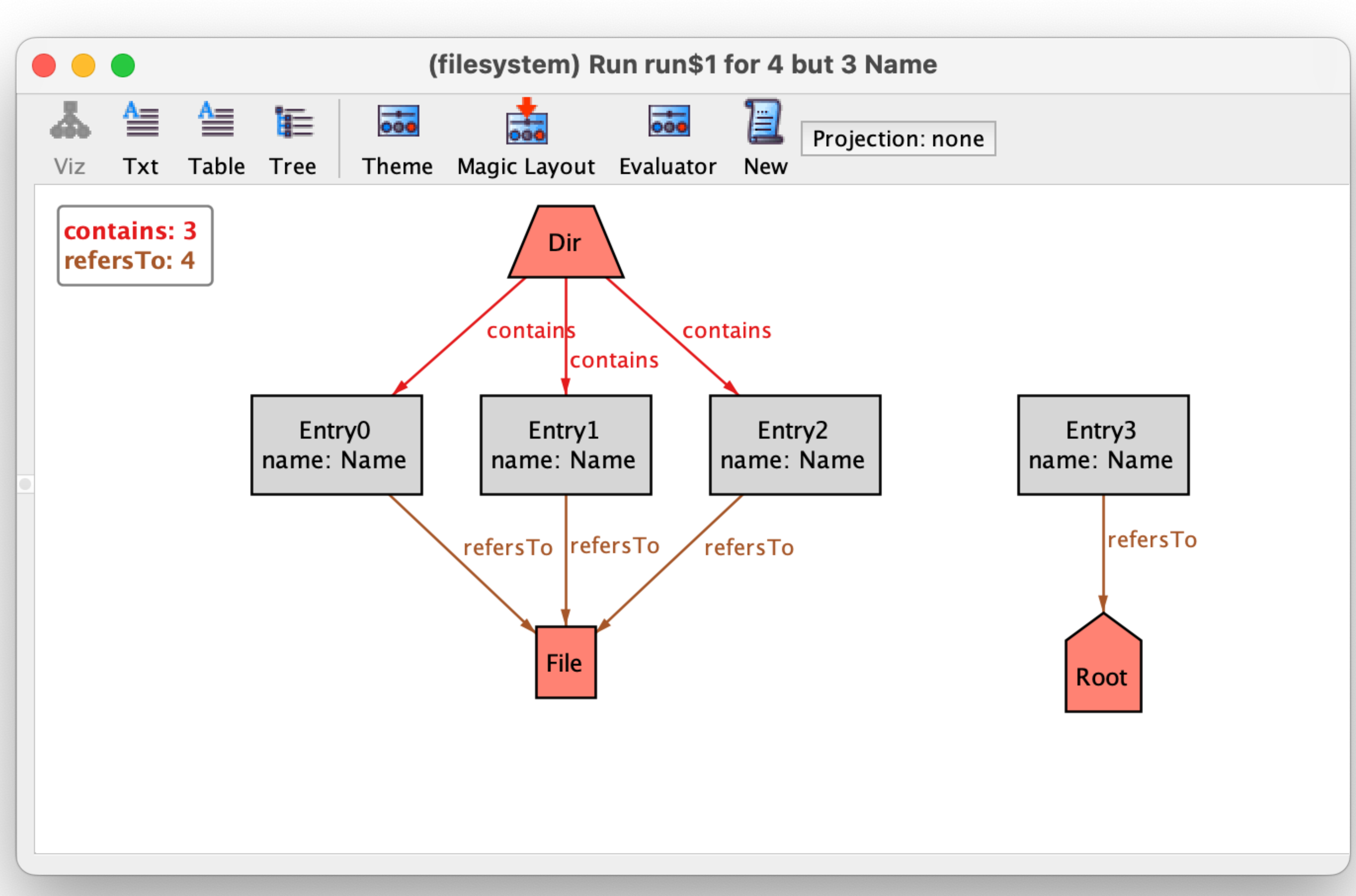


Themes

- The visualizer theme can be customised
- Customization can ease the understanding and help validate the model
- It is possible to customize colors, shapes, visibility, …

Theme customization

Theme customization



The missing constraints in FOL

fact {

// Each entry is contained in one directory

$\forall e : \texttt{Entry} \,.\, \exists o : \texttt{Object} \,.\, \texttt{contains}(o, e)$

$\forall e : \texttt{Entry}, o_1, o_2 : \texttt{Object} \,.\, \texttt{contains}(o_1, e) \wedge \texttt{contains}(o_2, e) \rightarrow o_1 = o_2$

// All objects except the root are referred to in at least one entry

$\forall o : \texttt{Object}(o) \,.\, \neg\texttt{Root}(o) \rightarrow \exists e : \texttt{Entry} \,.\, \texttt{refersTo}(e, o)$

$\forall e : \texttt{Entry}, r : \texttt{Root} \,.\, \neg\texttt{refersTo}(e, r)$

// All directories are referred to in at most one entry

$\forall d : \texttt{Dir}, e_1, e_2 : \texttt{Entry} \,.\, \texttt{refersTo}(e_1, d) \wedge \texttt{refersTo}(e_2, d) \rightarrow e_1 = e_2$

// Different entries in a directory must have different names

$\forall d : \texttt{Dir}, n : \texttt{Name}, e_1, e_2 : \texttt{Entry} \,.\, \texttt{contains}(d, e_1) \wedge \texttt{contains}(d, e_2) \wedge \texttt{name}(e_1, n) \wedge \texttt{name}(e_2, n) \rightarrow e_1 = e_2$

}

Logical operators

`!` ϕ	$\neg\phi$
ϕ `&&` ψ	$\phi \land \psi$
ϕ `||` ψ	$\phi \lor \psi$
ϕ `=>` ψ	$\phi \to \psi$
ϕ `=>` ψ `else` θ	$(\phi \land \psi) \lor (\neg\phi \land \theta)$
ϕ `<=>` ψ	$\phi \leftrightarrow \psi$

Logical operators

`not` ϕ	$\neg\phi$
ϕ `and` ψ	$\phi \wedge \psi$
ϕ `or` ψ	$\phi \vee \psi$
ϕ `implies` ψ	$\phi \rightarrow \psi$
ϕ `implies` ψ `else` θ	$(\phi \wedge \psi) \vee (\neg\phi \wedge \theta)$
ϕ `iff` ψ	$\phi \leftrightarrow \psi$

Quantifiers

`all` x **`:`** A | ϕ $\qquad$ $\forall x : A \,.\, \phi$

`some` x **`:`** A | ϕ $\qquad$ $\exists x : A \,.\, \phi$

Atomic formulas

x `=` y	$x = y$
x `!=` y	$x \neq y$
x_1`->`$\cdots$`->`x_n **`in`** R	$R(x_1, \ldots, x_n)$
x_1`->`$\cdots$`->`x_n **`not in`** R	$\neg R(x_1, \ldots, x_n)$

The missing constraints in Alloy

```
fact {
	// Each entry is contained in one directory
	all e : Entry | some o : Object | o->e in contains
	all e : Entry, o1,o2 : Object | o1->e in contains and o2->e in contains implies o1 = o2

	// All objects except the root are referred to in at least one entry
	all o : Object | o not in Root implies some e : Entry | e->o in refersTo
	all e : Entry, r : Root | e->r not in refersTo

	// All directories are referred to in at most one entry
	all d : Dir, e1,e2 : Entry | e1->d in refersTo and e2->d in refersTo implies e1 = e2

	// Different entries in a directory must have different names
	all d : Dir, n : Name, e1,e2 : Entry | d->e1 in contains and d->e2 in contains and e1->n in name and e2->n in name implies e1 = e2
}
```

Relational logic

Relational logic

- *Relational logic* extends FOL
- Adds new atomic formulas, namely cardinality checks
- Adds operators to combine predicates (relations) into more complex predicates
- Adds transitive and reflexive closures, which cannot be expressed in FOL

Atomic formulas

`// Subset test`		
R **`in`** S	$R \subseteq S$	$\forall x_1, \ldots, x_n \cdot R(x_1, \ldots, x_n) \rightarrow S(x_1, \ldots, x_n)$
R **`not in`** S	$R \nsubseteq S$	
`// Equality test`		
R `=` S	$R = S$	$\forall x_1, \ldots, x_n \cdot R(x_1, \ldots, x_n) \leftrightarrow S(x_1, \ldots, x_n)$
R **`!=`** S	$R \neq S$	

Atomic formulas

```
// Cardinality tests
```

some R	$\mid R \mid > 0$	$\exists x_1, \ldots, x_n \,.\, R(x_1, \ldots, x_n)$
no R	$\mid R \mid = 0$	$\forall x_1, \ldots, x_n \,.\, \neg R(x_1, \ldots, x_n)$
lone R	$\mid R \mid < 2$	$\forall x_1, \ldots, x_n, x'_1, \ldots, x'_n \,.\, R(x_1, \ldots, x_n) \land R(x'_1, \ldots, x'_n) \rightarrow x_1 = x'_1 \land \ldots \land x_n = x'_n$
one R	$\mid R \mid = 1$	

Set operators

```
// Union
```

$R + S$ $\quad R \cup S \quad (R + S)(x_1, \ldots, x_n) \leftrightarrow R(x_1, \ldots, x_n) \vee S(x_1, \ldots, x_n)$

```
// Intersection
```

$R \;\&\; S$ $\quad R \cap S \quad (R \;\&\; S)(x_1, \ldots, x_n) \leftrightarrow R(x_1, \ldots, x_n) \wedge S(x_1, \ldots, x_n)$

```
// Difference
```

$R - S$ $\quad R \backslash S \quad (R - S)(x_1, \ldots, x_n) \leftrightarrow R(x_1, \ldots, x_n) \wedge \neg S(x_1, \ldots, x_n)$

Relational constants

```
// Universe
```

`univ` $\forall x \,.\, \mathbf{univ}(x)$

```
// Empty set
```

`none` $\forall x \,.\, \neg \mathbf{none}(x)$

```
// Identity
```

`iden` $\forall x_1, x_2 \,.\, \mathbf{iden}(x_1, x_2) \leftrightarrow x_1 = x_2$

Quantifiers

all x **:** A $\mid$ ϕ	$\forall x \,.\, x \in A \rightarrow \phi$
some x **:** A $\mid$ ϕ	$\exists x \,.\, x \in A \wedge \phi$
all disj x**,**y **:** A $\mid$ ϕ	$\forall x, y \,.\, x \in A \wedge y \in A \wedge x \neq y \rightarrow \phi$
some disj x**,**y **:** A $\mid$ ϕ	$\exists x, y \,.\, x \in A \wedge y \in A \wedge x \neq y \wedge \phi$

Relational operators

```
// Cartesian product
```

R `->` S $R \times S$ $(R \texttt{ -> } S)(x_1, \ldots, x_n, y_1, \ldots, y_m) \leftrightarrow R(x_1, \ldots, x_n) \land S(y_1, \ldots, y_m)$

```
// Transpose or converse
```

$\sim R$ R° $(\sim R)(x_1, x_2) \leftrightarrow R(x_2, x_1)$

```
// Range restriction
```

R `:>` A $(R \texttt{ :> } A)(x_1, \ldots, x_n) \leftrightarrow R(x_1, \ldots, x_n) \land A(x_n)$

```
// Domain restriction
```

A `<:` R $(A \texttt{ <: } R)(x_1, \ldots, x_n) \leftrightarrow R(x_1, \ldots, x_n) \land A(x_1)$

Inclusion vs subset

```
all x : Dir | x in Object
```

~~$\forall x : \texttt{Dir}.(x) \in \texttt{Object}$~~

$\forall x : \texttt{Dir}.\{(x)\} \subseteq \texttt{Object}$

Inclusion vs subset

```
all x : Entry | some y : Name | x->y in name
```

~~$\forall x : \texttt{Entry} \,.\, \exists y : \texttt{Name} \,.\, (x, y) \in \texttt{name}$~~

$\forall x : \texttt{Entry} \,.\, \exists y : \texttt{Name} \,.\, \{(x)\} \times \{(y)\} \subseteq \texttt{name}$

$\forall x : \texttt{Entry} \,.\, \exists y : \texttt{Name} \,.\, \{(x, y)\} \subseteq \texttt{name}$

Composition

$$R \,\bullet\, S$$

$$S \circ R$$

$$(R \,\bullet\, S)(x, y) \leftrightarrow \exists z \,.\, R(x, z) \wedge S(z, y)$$

$$(R \,\bullet\, S)(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m) \leftrightarrow \exists z \cdot R(x_1, \ldots, x_{n-1}, z) \wedge S(z, y_2, \ldots, y_m)$$

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

```
contains . name
```

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

contains . name	
Root	Name0

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

contains . name	
Root	Name0
Root	Name1

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

contains . name	
Root	Name0
Root	Name1
Dir	Name1

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

contains . name	
Root	Name0
Root	Name1
Dir	Name1

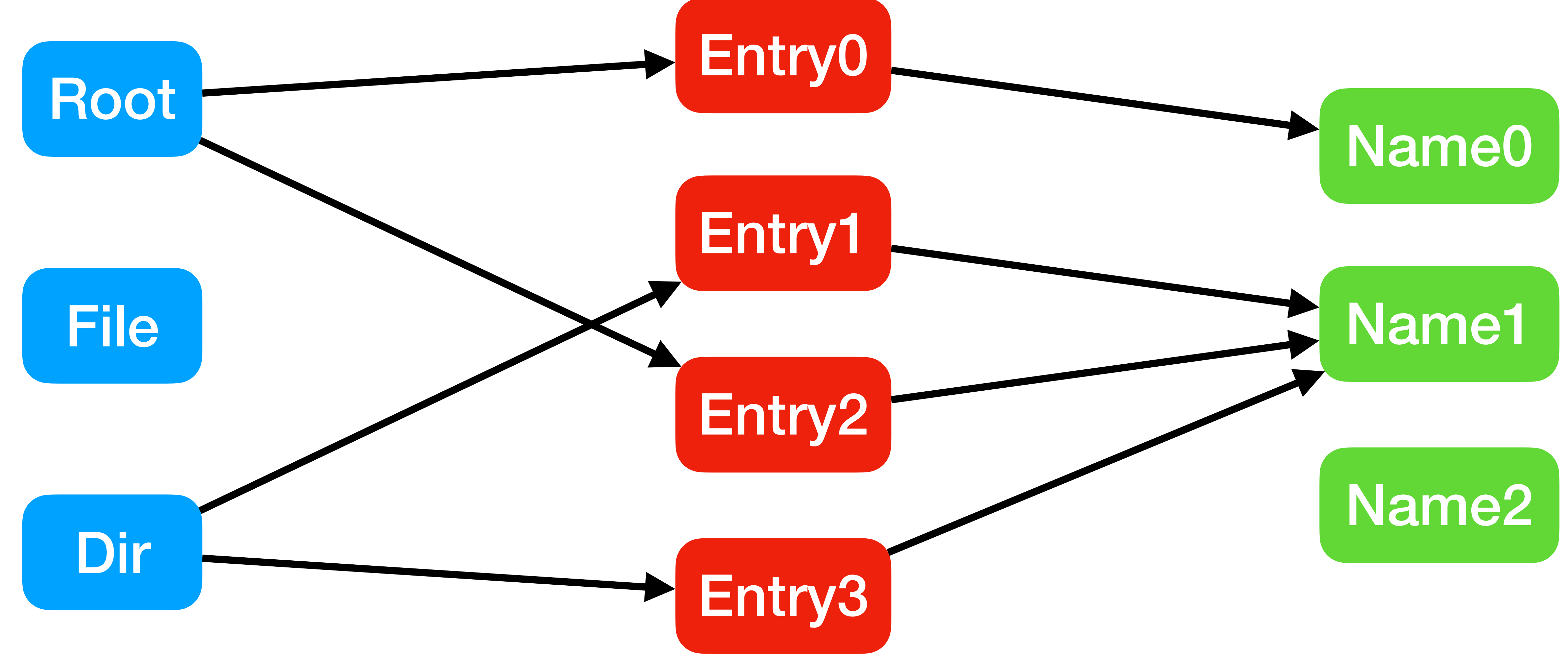
Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

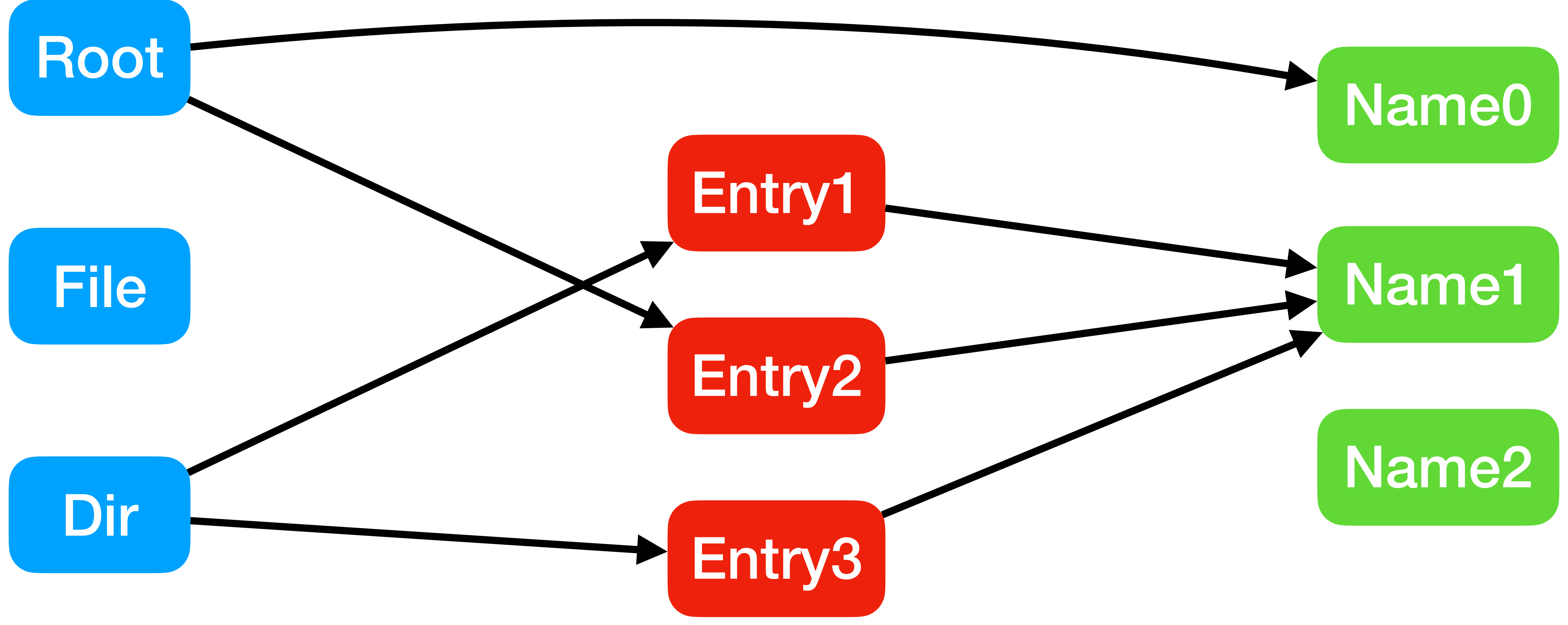
name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

contains . name	
Root	Name0
Root	Name1
Dir	Name1

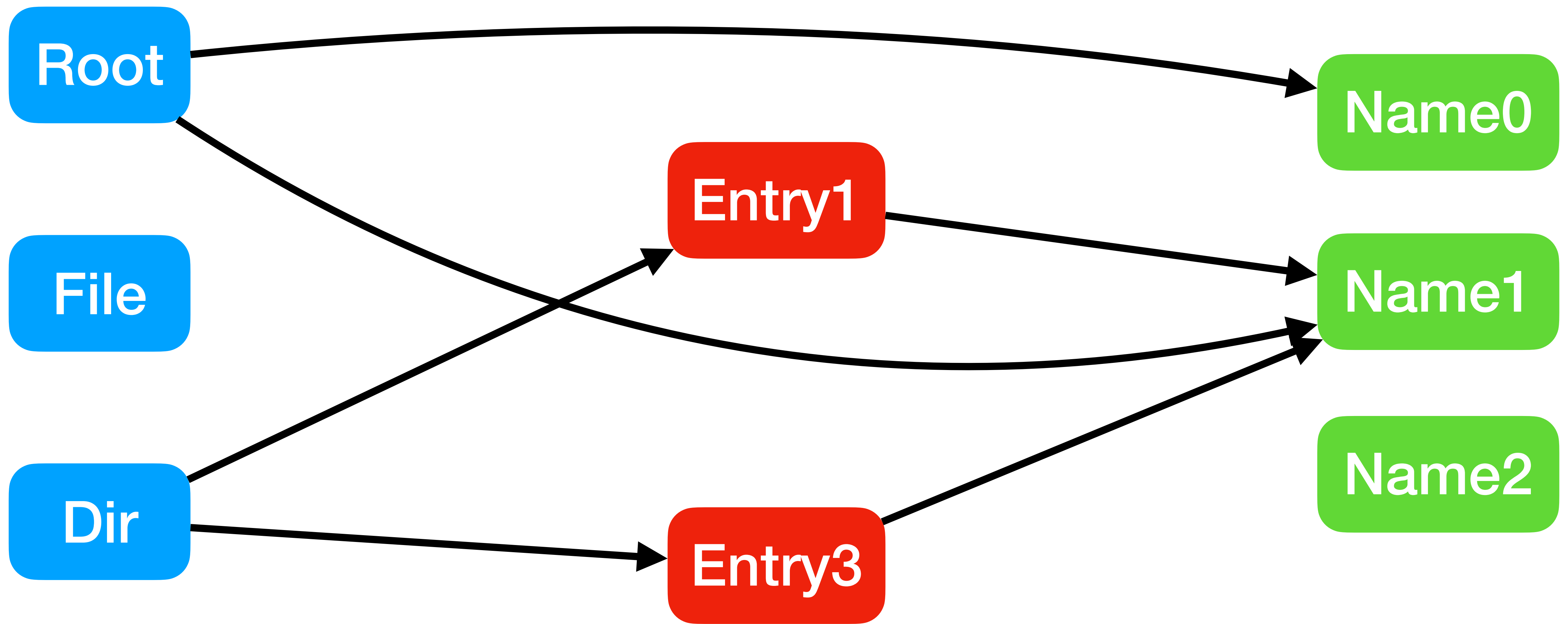
Composition



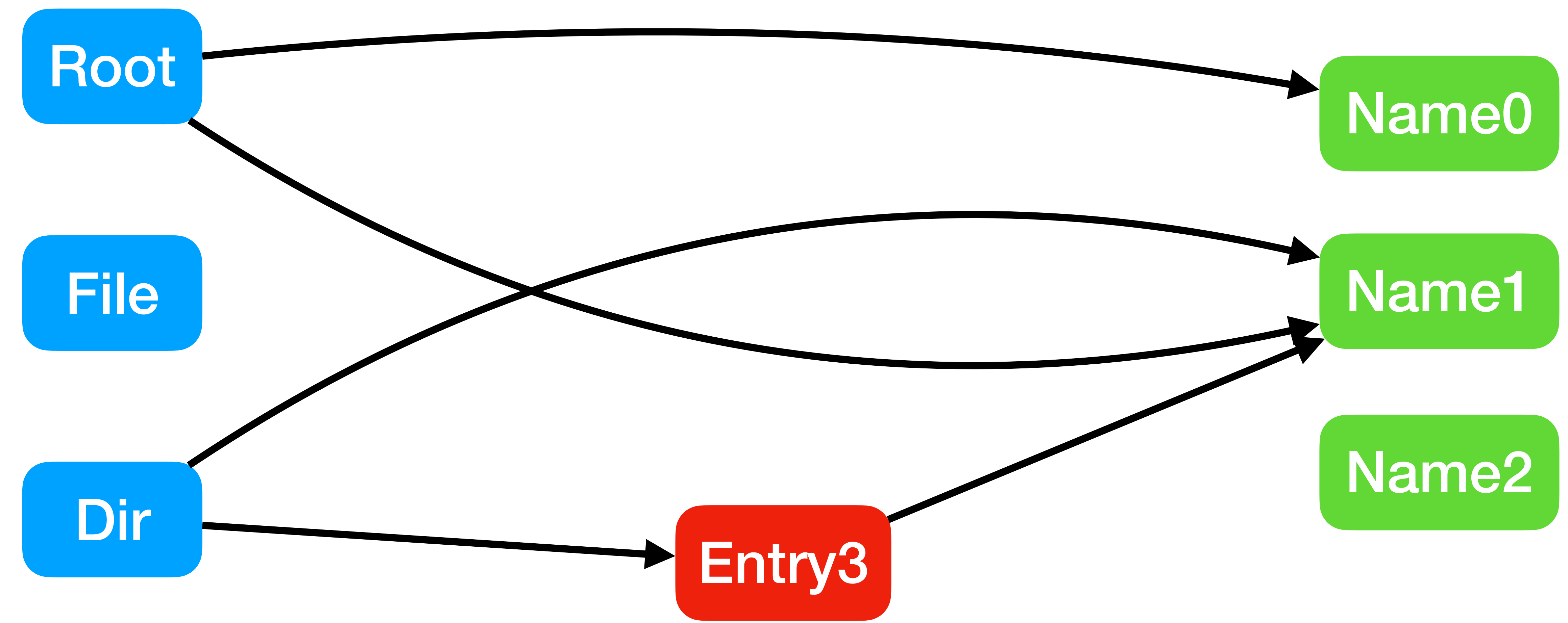
Composition



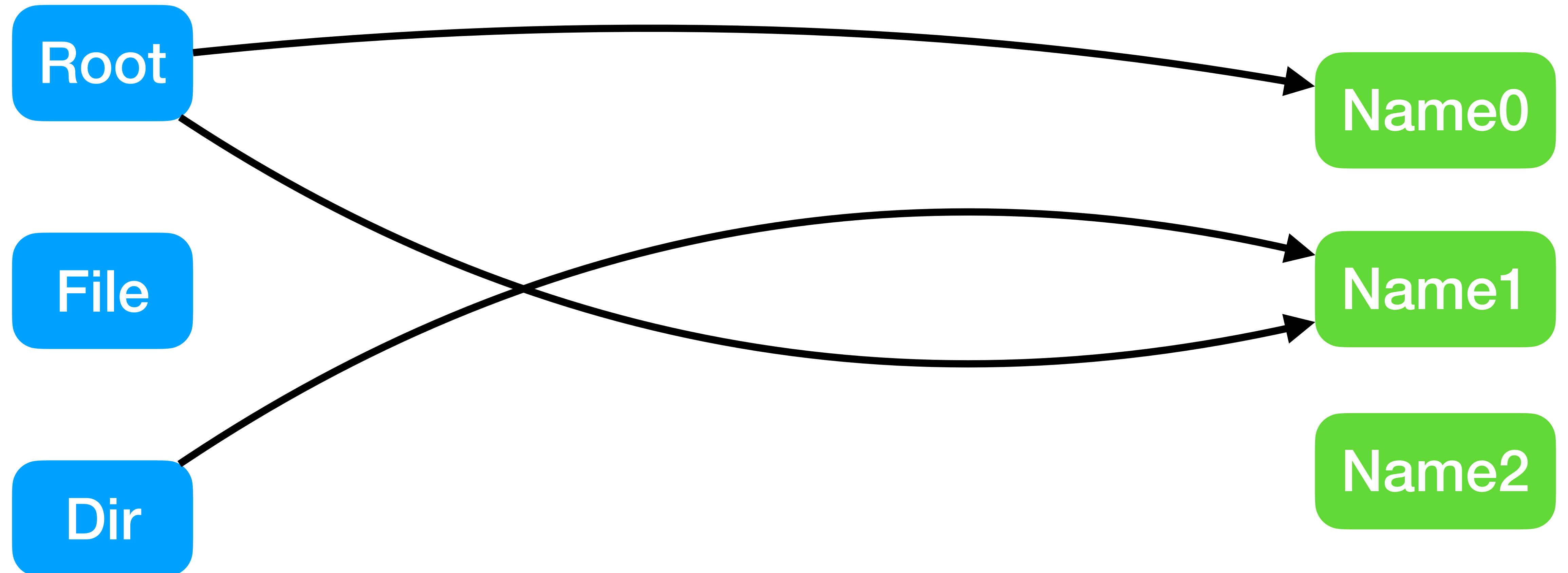
Composition



Composition



Composition



Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

contains . Entry

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

Entry
Entry0
Entry1
Entry2
Entry3

contains . Entry
Root

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

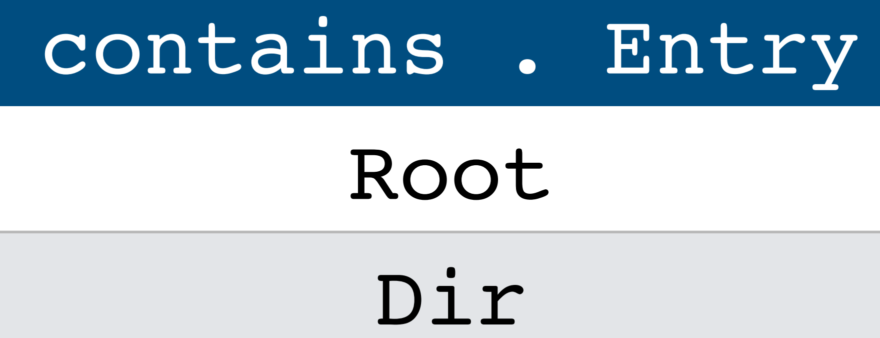
Entry
Entry0
Entry1
Entry2
Entry3

contains . Entry
Root

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3



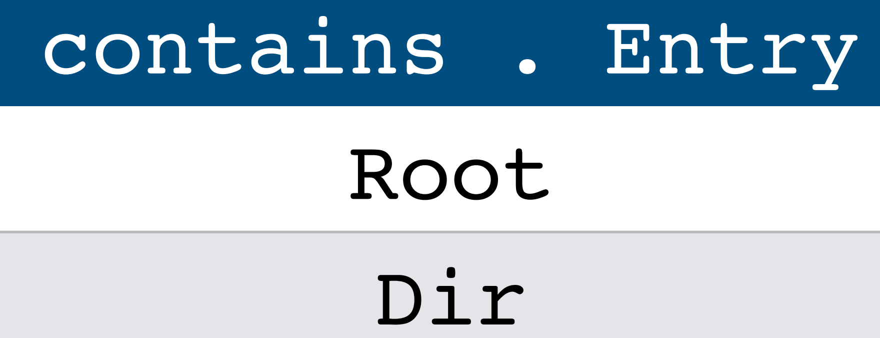
contains . Entry
Root
Dir

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

Entry
Entry0
Entry1
Entry2
Entry3



contains . Entry
Root
Dir

Composition

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

contains . Entry
Root
Dir

Composition

Root
Root

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

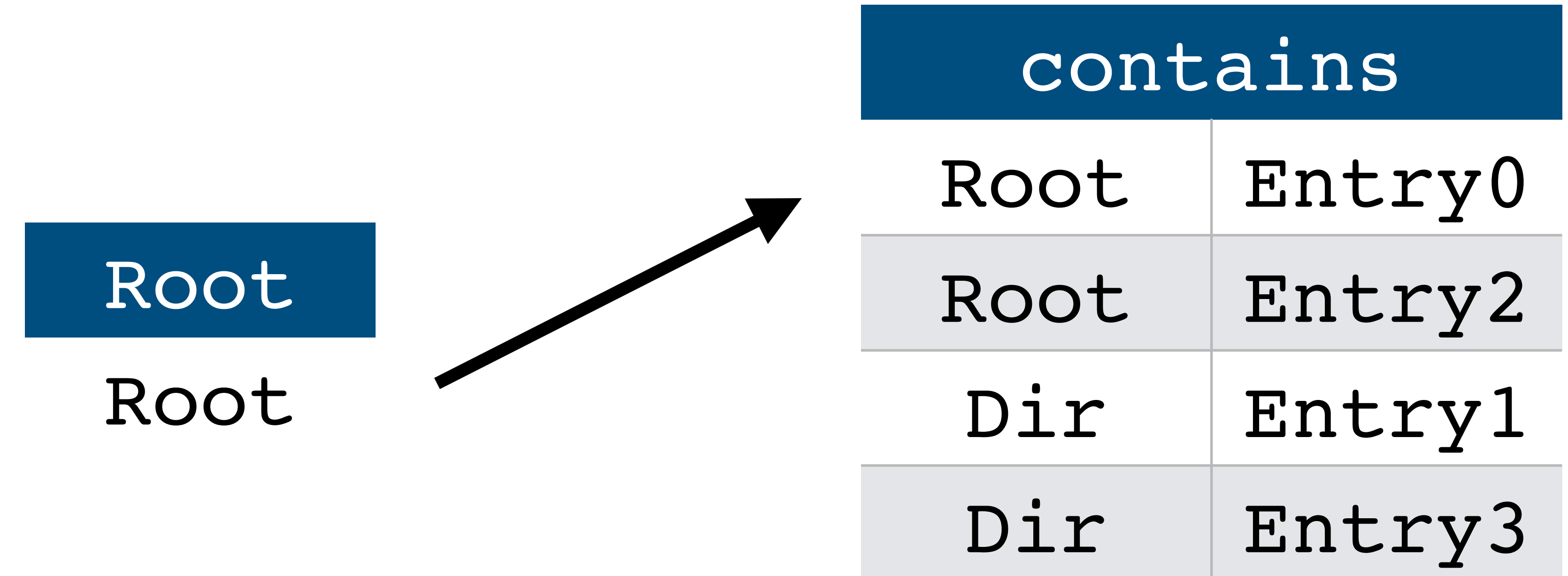
Composition

Root
Root

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

`Root . contains`

Composition

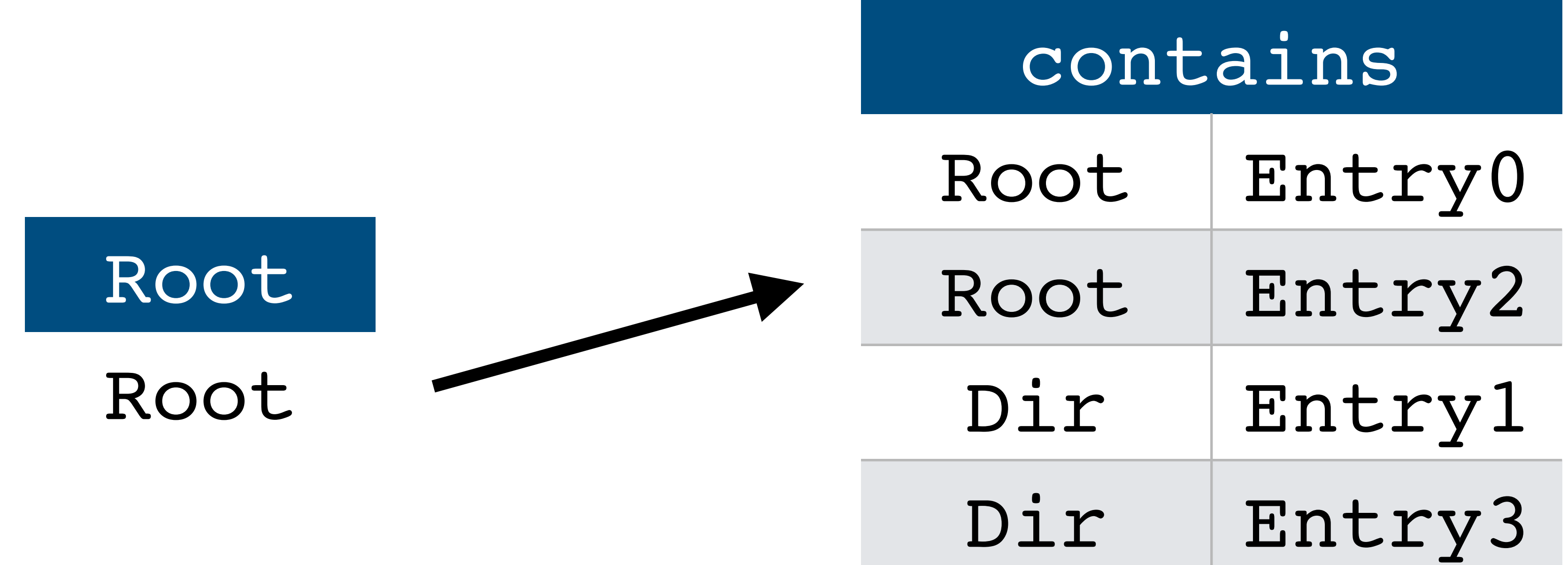


Root
Root

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Root . contains
Entry0

Composition



Root . contains
Entry0
Entry2

Composition

Root
Root

contains	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Root . contains
Entry0
Entry2

From FOL to RL

From FOL to RL

```
// All objects except the root are referred to in at least one entry
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all e : Entry, r : Root | e->r not in refersTo
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all e : Entry, r : Root | e->r not in refersTo

all e : Entry | e->Root not in refersTo
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all e : Entry, r : Root | e->r not in refersTo

all e : Entry | e->Root not in refersTo

all e : Entry | e not in refersTo.Root
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all e : Entry, r : Root | e->r not in refersTo

all e : Entry | e->Root not in refersTo

all e : Entry | e not in refersTo.Root

no refersTo.Root
```

From FOL to RL

From FOL to RL

```
// All objects except the root are referred to in at least one entry
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all o : Object | o not in Root implies some e : Entry | e->o in refersTo
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all o : Object | o not in Root implies some e : Entry | e->o in refersTo

all o : Object | o not in Root implies some refersTo.o
```

From FOL to RL

```
// All objects except the root are referred to in at least one entry

all o : Object | o not in Root implies some e : Entry | e->o in refersTo

all o : Object | o not in Root implies some refersTo.o

all o : Object-Root | some refersTo.o
```

From FOL to RL

```
// Different entries in a directory must have different names

all d:Dir, n:Name, e1,e2:Entry | d->e1 in contains and d->e2 in contains and e1->n in name and e2->n in name implies e1 = e2

all d:Dir, n:Name, e1,e2:Entry | e1 in d.contains and e2 in d.contains and e1 in name.n and e2 in name.n implies e1 = e2

all d:Dir, n:Name, e1,e2:Entry | e1 in d.contains and e1 in name.n and e2 in d.contains and e2 in name.n implies e1 = e2

all d:Dir, n:Name, e1,e2:Entry | e1 in (d.entries & name.n) and e2 in (d.entries & name.n) implies e1 = e2

all d : Dir, n : Name | lone (d.entries & name.n)
```

The missing constraints in Alloy

```
fact {
   // Each entry is contained in one directory
   all e : Entry | one contains.e

   // All objects except the root are referred to in at least one entry
   all o : Object - Root | some refersTo.o
   no refersTo.Root

   // All directories are referred to in at most one entry
   all d : Dir | lone refersTo.d

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

A question of style

A question of style

```
// First order style
all x,y : Entry, n : Name | x->n in name and y->n in name implies x=y
```

A question of style

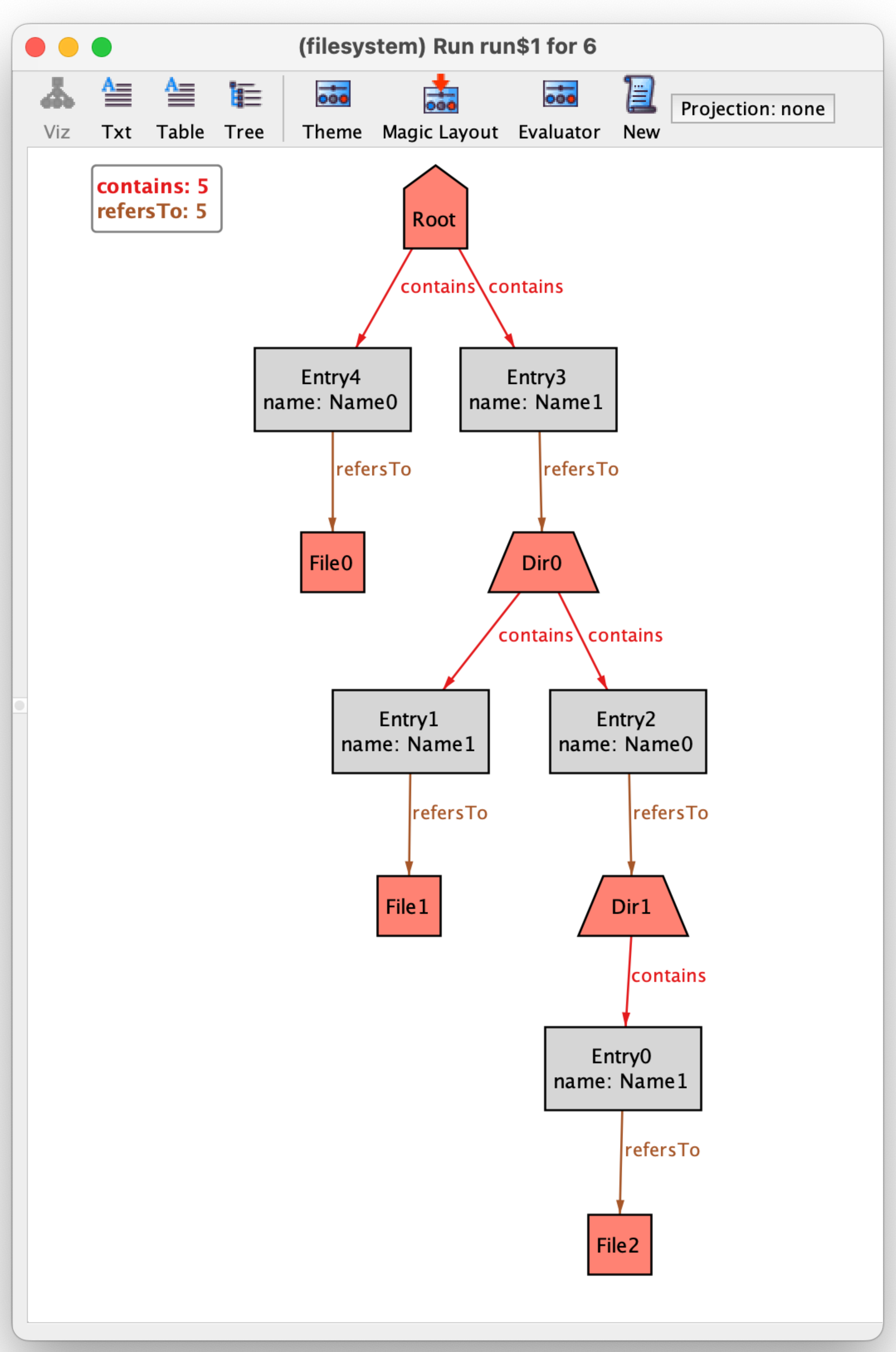
```
// First order style
all x,y : Entry, n : Name | x->n in name and y->n in name implies x=y

// Relational or navigational style
all n : Name | lone name.n
```

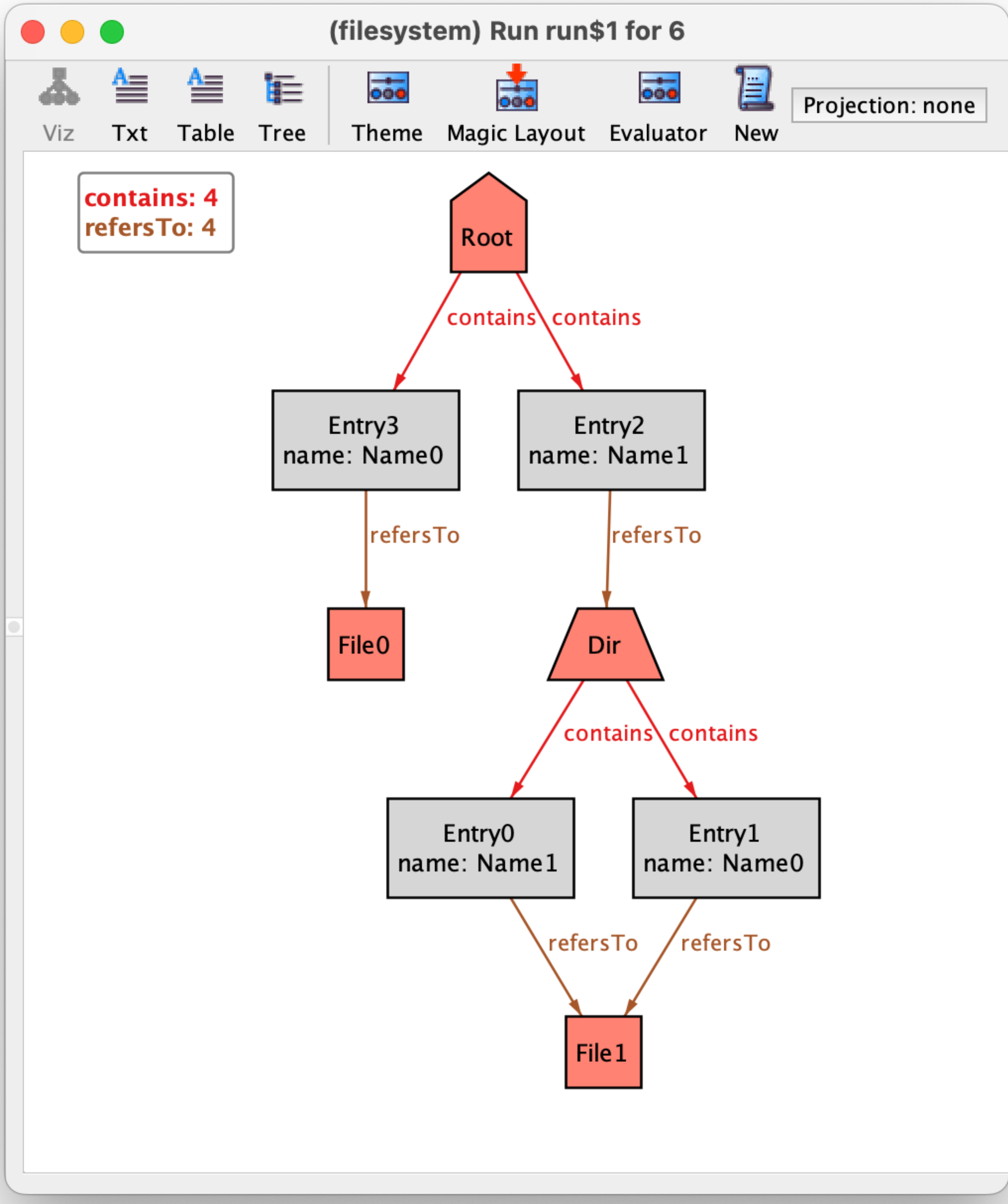
A question of style

```
// First order style
all x,y : Entry, n : Name | x->n in name and y->n in name implies x=y

// Relational or navigational style
all n : Name | lone name.n

// Point-free style
name.~name in iden
```

Verification

Some instances

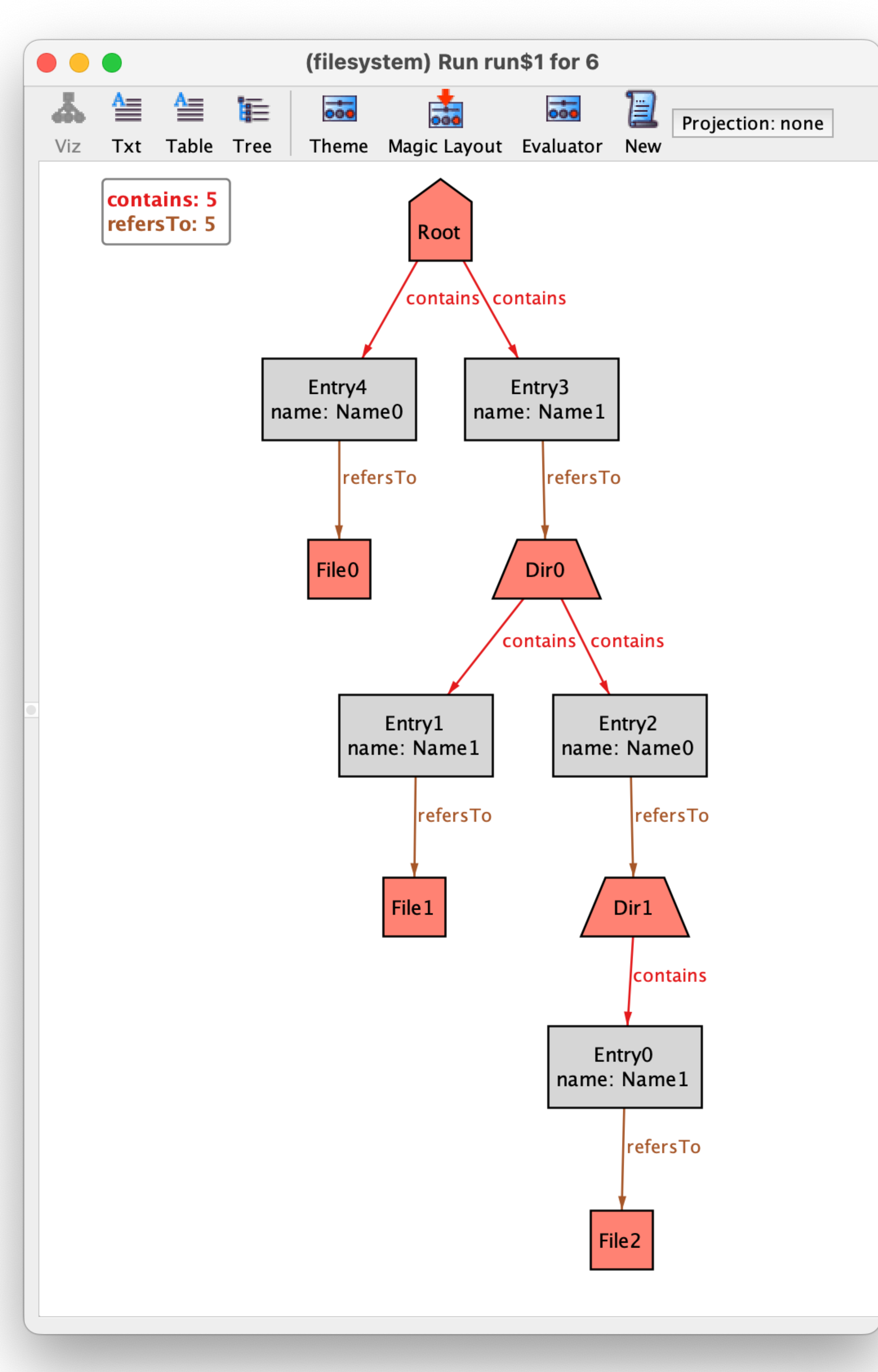




A desirable assertion

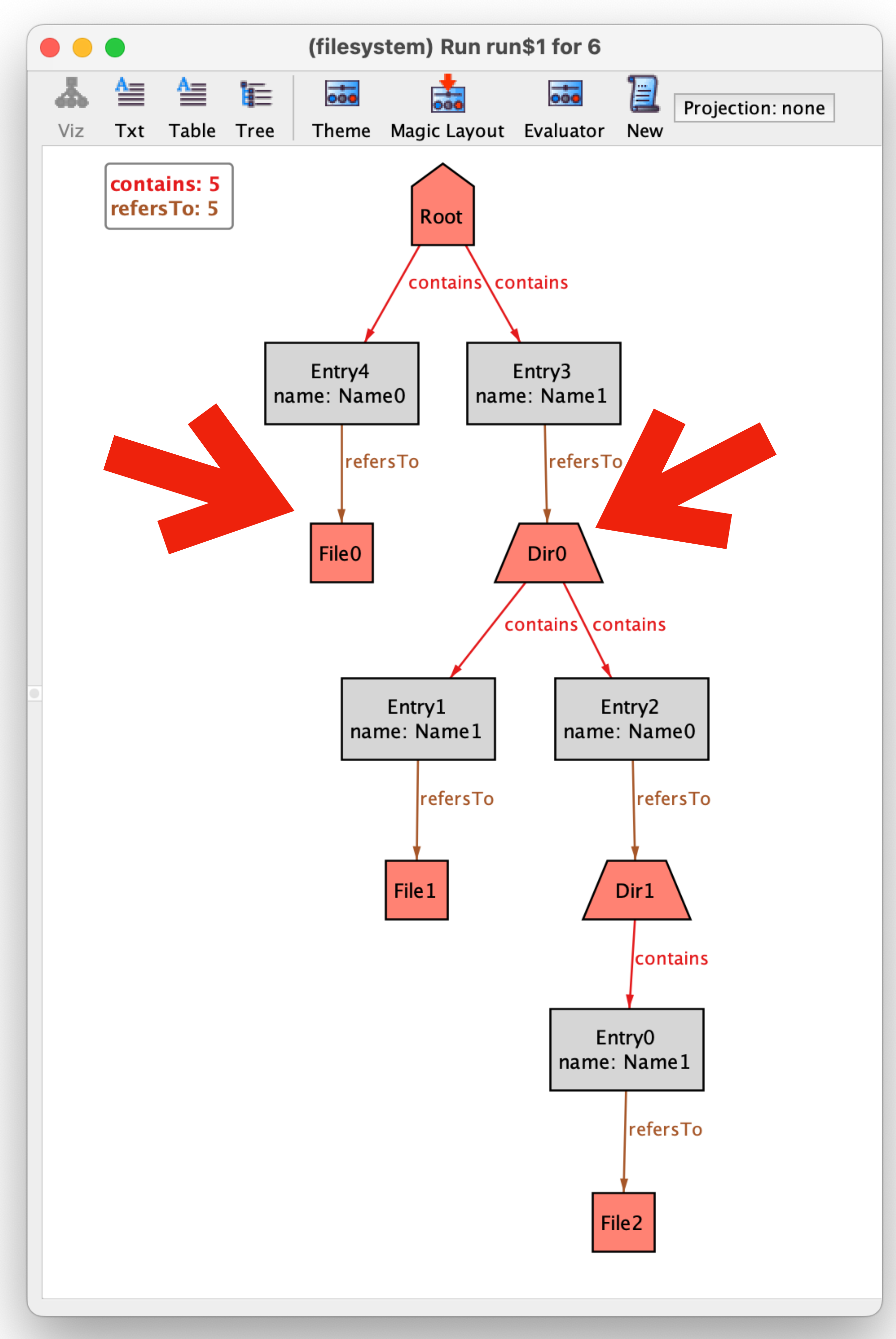
```
assert NoPartitions {
  // All objects are reachable from the root
  ???
}

check NoPartitions
```

Reachable objects



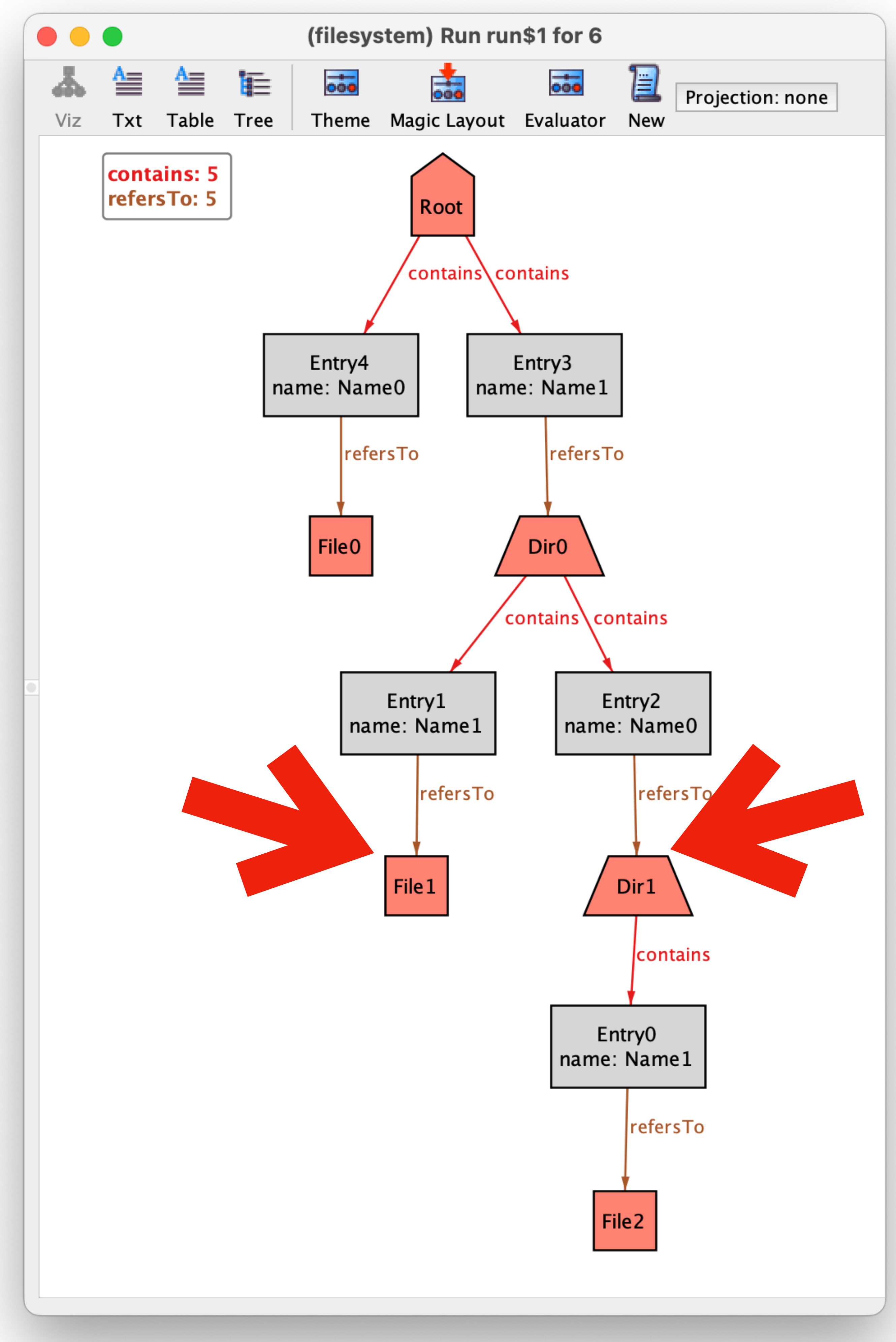
Reachable objects

`Root.contains.refersTo`



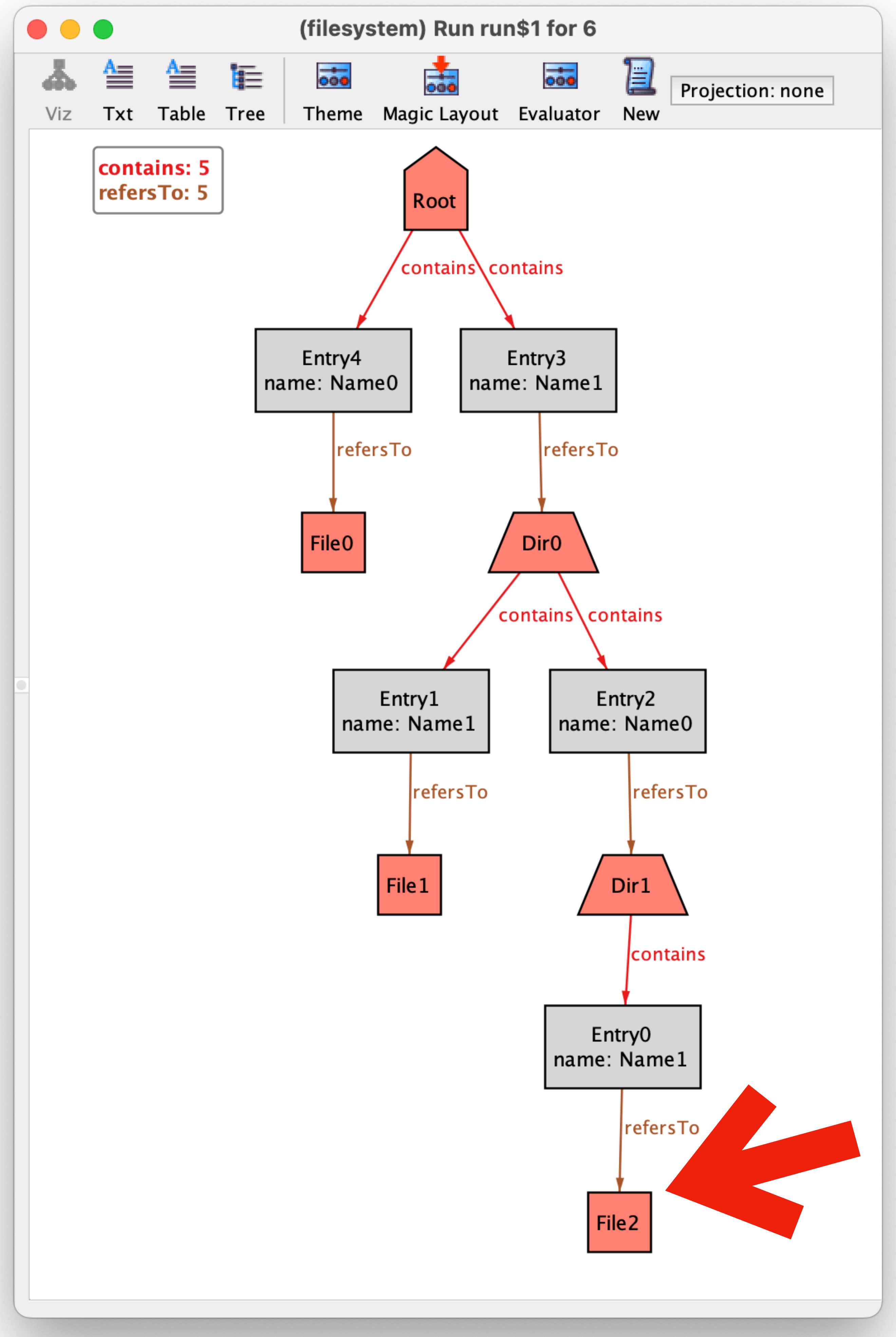
Reachable objects

`Root.contains.refersTo.contains.refersTo`



Reachable objects

`Root.contains.refersTo.contains.refersTo.contains.refersTo`



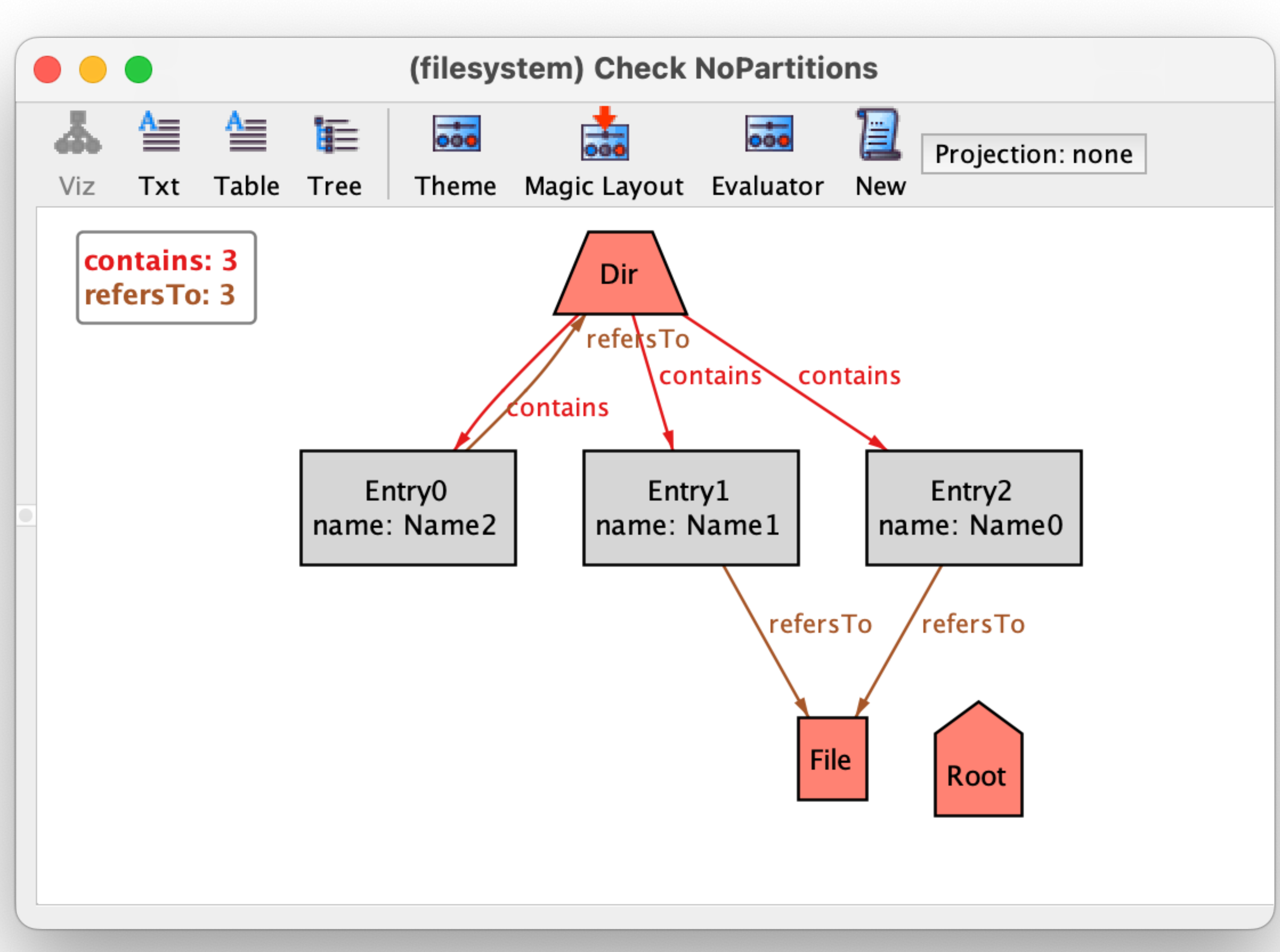
Closures

```
// Transitive closure
^R = R + R.R + R.R.R + R.R.R.R + …

// Reflexive transitive closure
*R = ^R + iden
```

A desirable assertion

```
assert NoPartitions {
  // All objects are reachable from the root
  Object in Root.*(contains.refersTo)
}

check NoPartitions
```

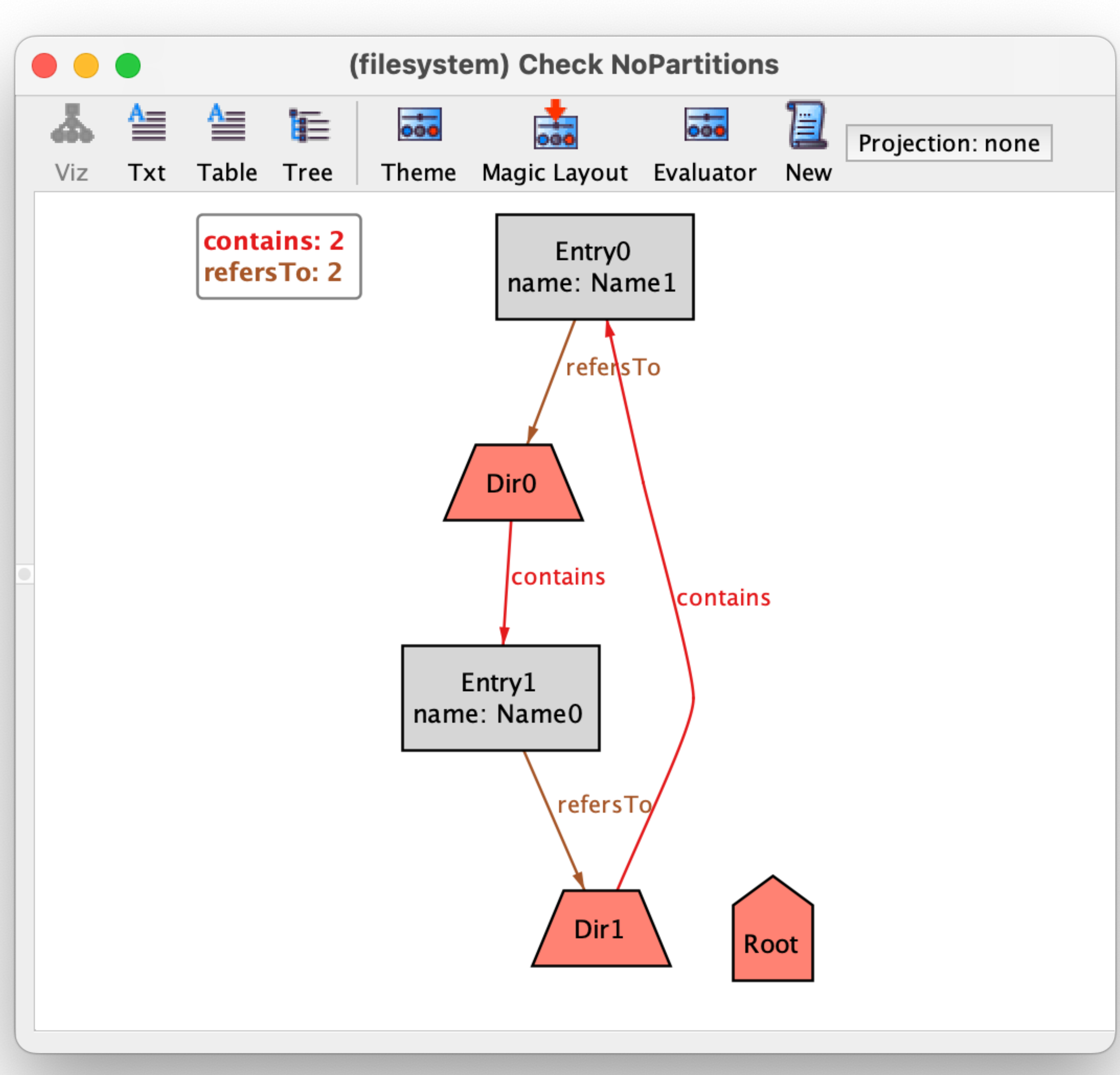
A counter-example



An unanticipated constraint

```
fact {
  // Each entry is contained in one directory
  all e : Entry | one contains.e

  // All objects except the root are referred to in at least one entry
  all o : Object - Root | some refersTo.o
  no refersTo.Root

  // All directories are referred to in at most one entry
  all d : Dir | lone refersTo.d

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

  // A directory cannot be contained in itself
  all d : Dir | d not in d.contains.refersTo
}
```

Another counter-example



An unanticipated constraint

```
fact {
  // Each entry is contained in one directory
  all e : Entry | one contains.e

  // All objects except the root are referred to in at least one entry
  all o : Object - Root | some refersTo.o
  no refersTo.Root

  // All directories are referred to in at most one entry
  all d : Dir | lone refersTo.d

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

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

```
Executing "Check NoPartitions"
   Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   586 vars. 37 primary vars. 860 clauses. 3ms.
   No counterexample found. Assertion may be valid. 2ms.
```

Increasing confidence

- Increase the scope of check commands

```
check NoPartitions for 6
```

- Use **`run`** commands to check consistency
- Namely, to check that specific scenarios are possible

Specifying scenarios

```
run Scenario1 {
   // An empty file system
   Object = Root
}

run Scenario2 {
   // A file system with only two files with different names
   some disj f1,f2 : File, disj e1,e2 : Entry, disj n1,n2 : Name {
      contains = Root->e1 + Root->e2
      refersTo = e1->f1 + e2->f2
      name     = e1->n1 + e2->n2
   }
}
```