Structural design with Alloy

Alcino Cunha

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

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

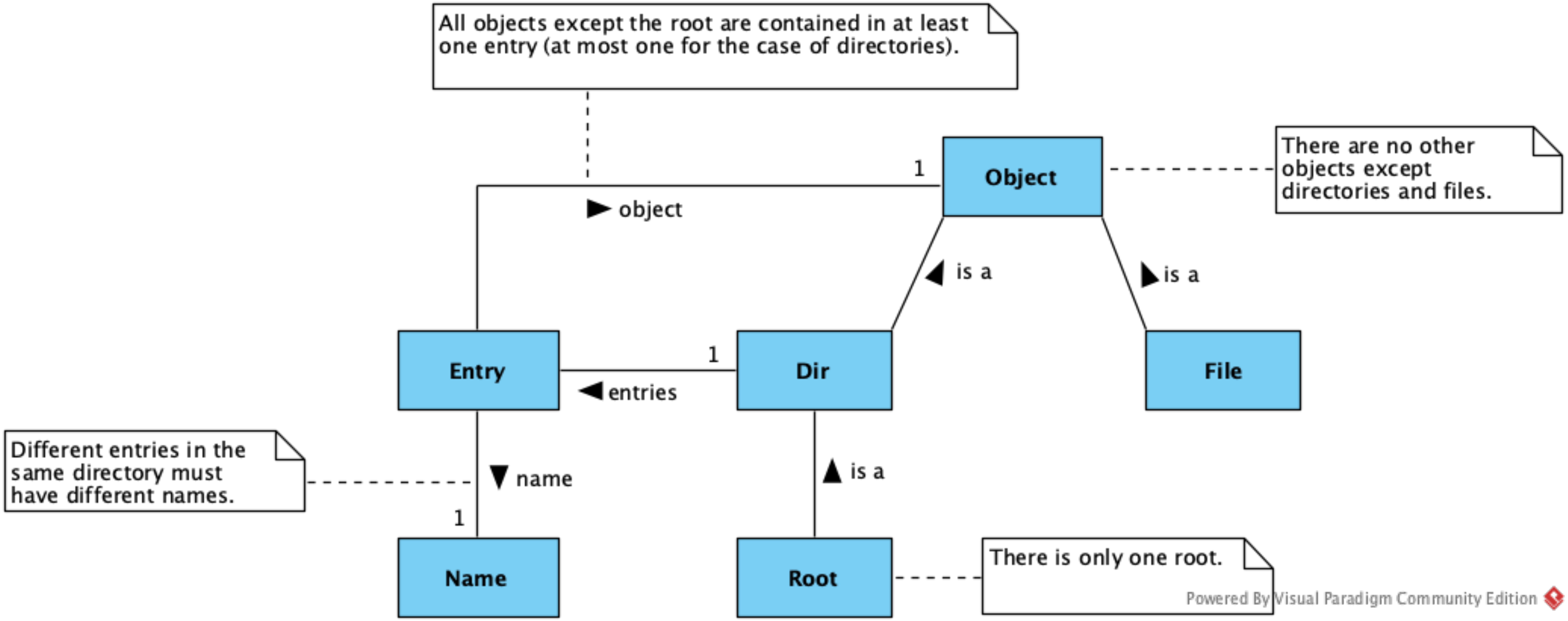
Software structures

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

Structural design

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

Domain modelling *a la* UML



Domain modelling *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?

Software design with Alloy

- Alloy is a formal modelling language
- Can be used to declare structures and events and specify constraints
- Models can be automatically analysed

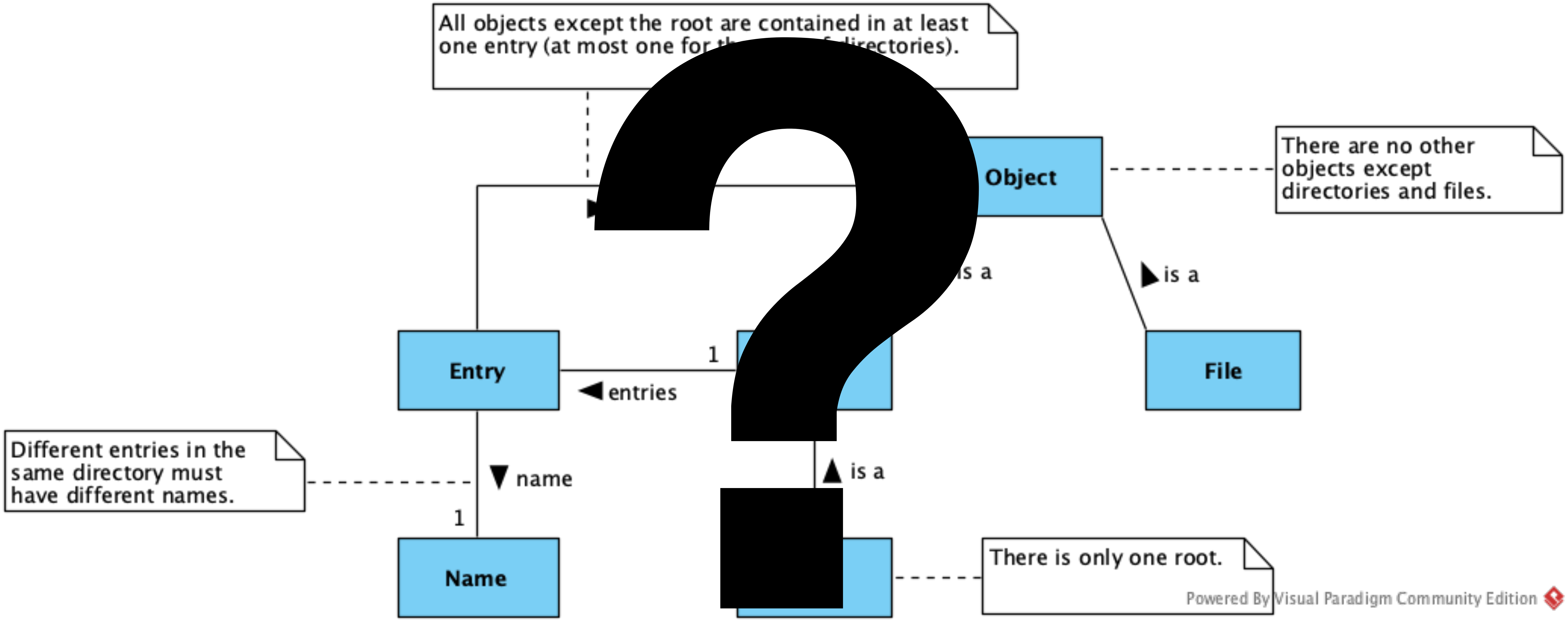
Software Abstractions

Logic, Language, and Analysis

Revised edition

Daniel Jackson

Domain modelling with Alloy



Signatures and fields

Entities = Signatures

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

“Is a” = Extension

```
sig Dir  extends Object {}
sig File extends Object {}
sig Root extends Dir {}
```

Signatures

- *Signatures* are sets
- Inhabited by *atoms* from a finite *universe* of discourse
- *Top-level* signatures are disjoint
- An *extension* signature is a subset of the *parent* signature
- Sibling extension signatures are disjoint

Relationships = Fields

```
sig Dir extends Object {
 entries : set Entry
}
sig Entry {
 object : 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*

Multiplicity constraints

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
}
```

Multiplicity constraints

`fact {` R `in` A m `->` m B `}`

Alloy	UML
set	0..*
lone	0..1
some	1..*
one	1

- In a multiplicity constraint the default multiplicity is **`set`**
- The target multiplicity can alternatively be specified in the declaration
- In field declarations the default target multiplicity is **`one`**

Multiplicity constraints

```
sig Dir extends Object {
 entries : set Entry
}
sig Entry {
 object : set Object,
 name   : set Name
}
fact Multiplicities {
 entries in Dir   one -> set Entry
 object  in Entry set -> one Object
 name    in Entry set -> one Name
}
```

Multiplicity constraints

```
sig Dir extends Object {
  entries : set Entry
}
sig Entry {
  object : one Object,
  name   : one Name
}
fact {
  entries in Dir one -> set Entry
}
```

Multiplicity constraints

```
sig Dir extends Object {
  entries : set Entry
}
sig Entry {
  object : Object,
  name   : Name
}
fact {
  entries in Dir one -> Entry
}
```

Bestiary

```
R in A set  -> some B       // R is entire
R in A set  -> lone B       // R is simple
R in A some -> set  B       // R is surjective
R in A lone -> set  B       // R is injective

R in A lone -> some B       // R is a representation
R in A some -> lone B       // R is an abstraction

R in A set  -> one  B       // R is a function
R in A lone -> one  B       // R is an injection
R in A some -> one  B       // R is a surjection

R in A one  -> one  B       // R is a bijection
```

Analysis

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 visualiser it 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 a 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

The small scope hypothesis

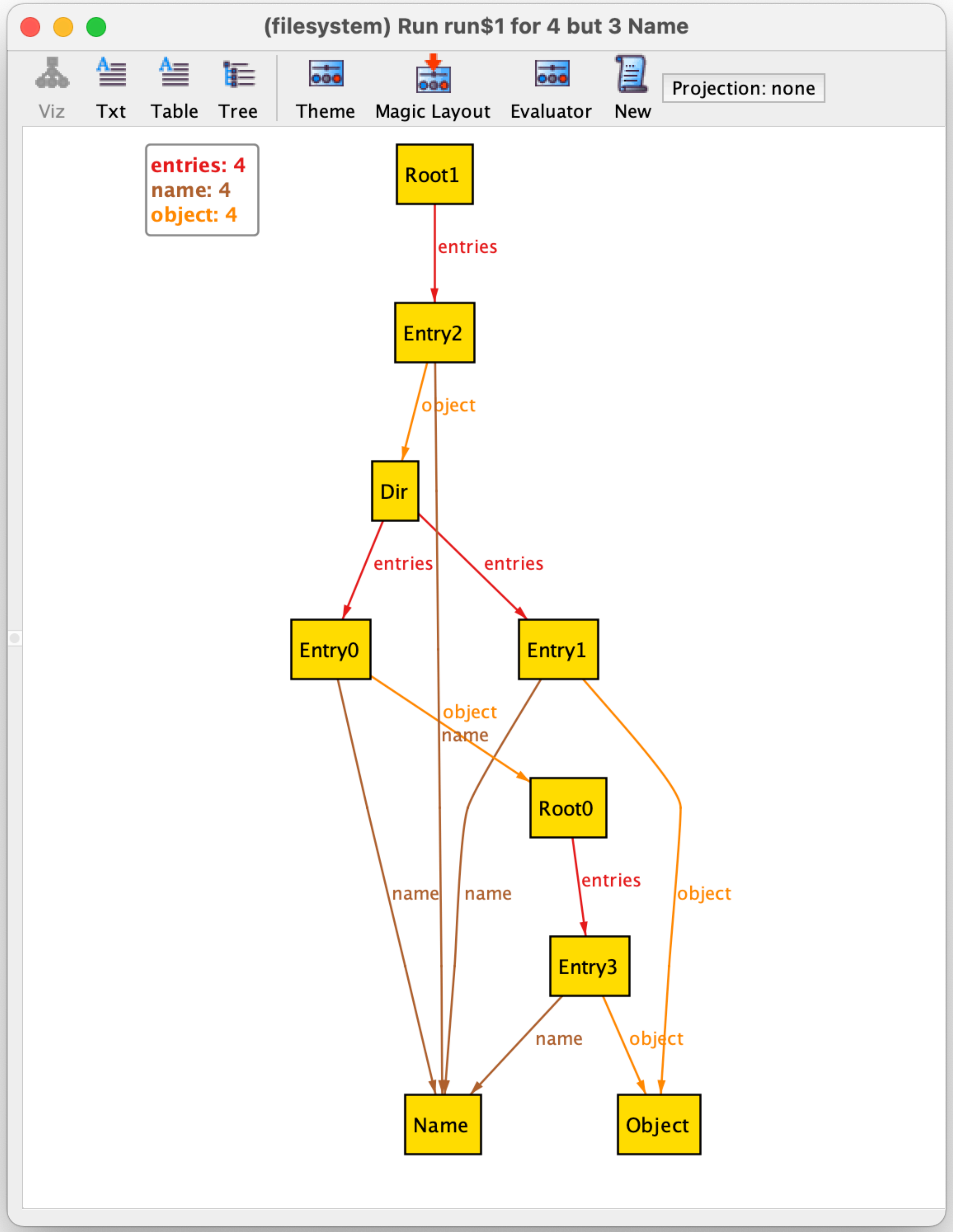
- If **`run { φ }`** returns an instance then φ is *consistent*, else φ **MAY** be *inconsistent*
    - Could be consistent with a bigger scope!
- If **`check { φ }`** returns an instance then φ is *invalid,* else φ **MAY** be *valid*
    - Could be invalid with a bigger scope!!!
- Anecdotical evidence 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
```

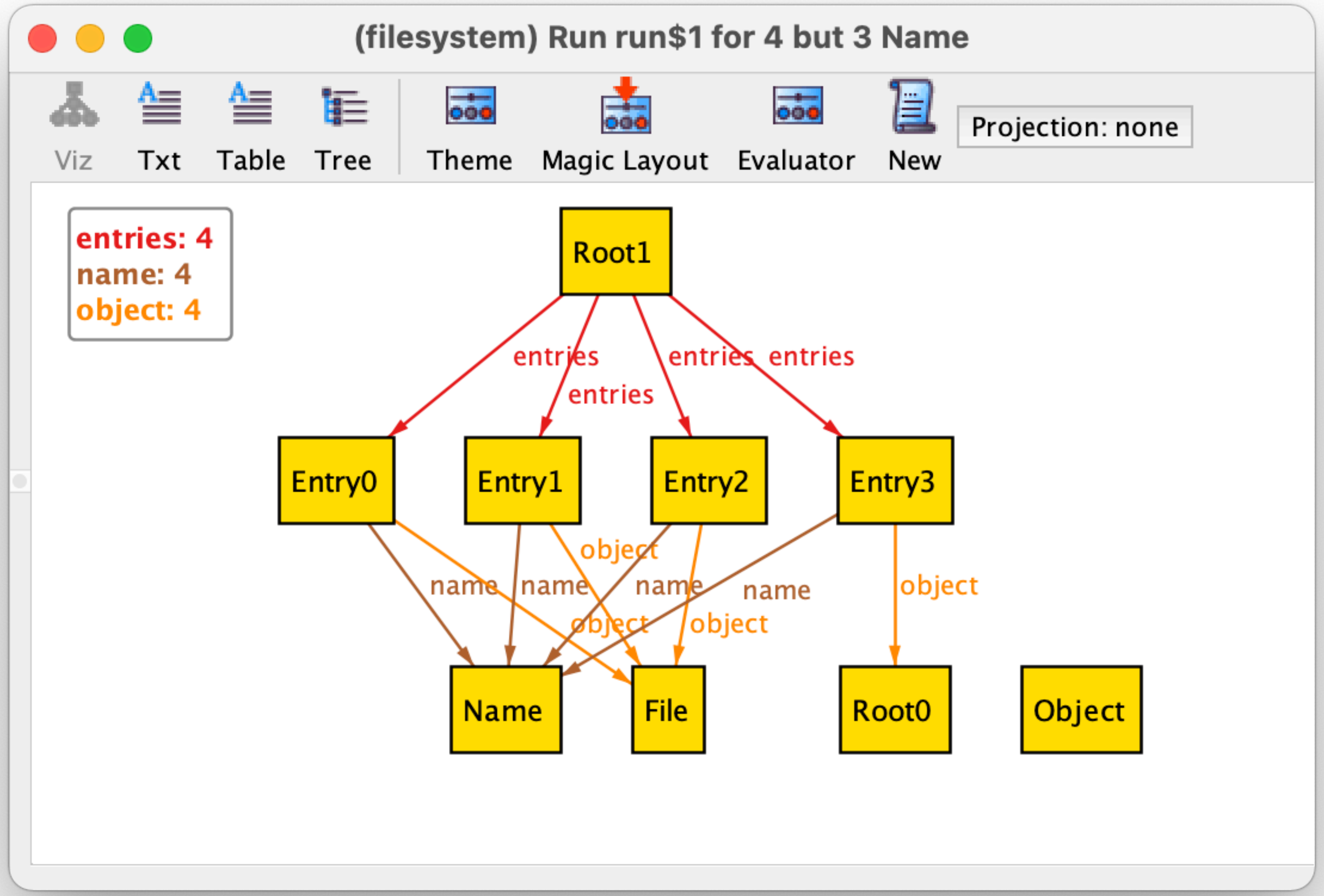
A simple command

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

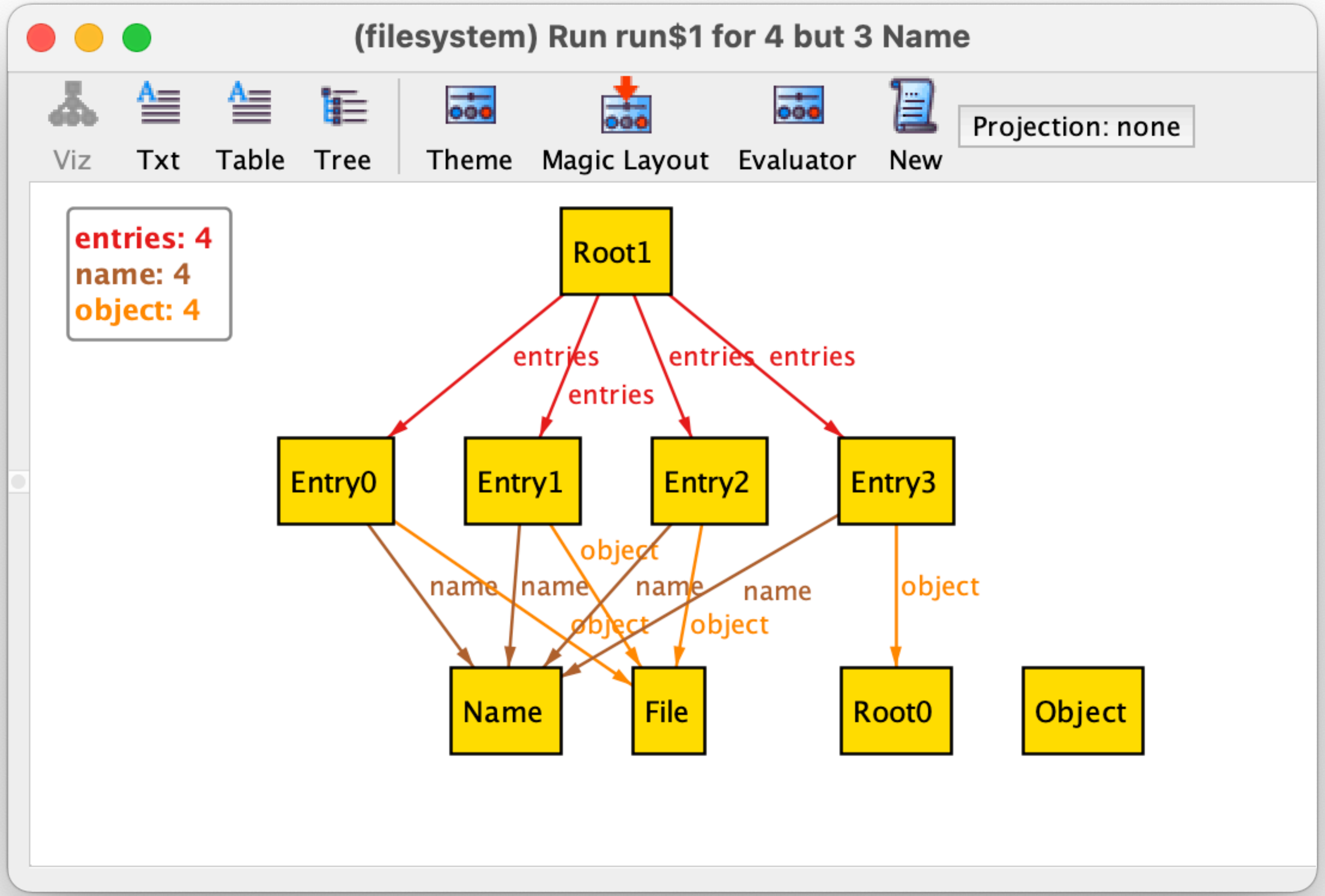


A simple command

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



Instances as graphs



Instances as sets

```
Object  = {(Object),(Root0),(Root1),(File)}
Dir     = {(Root0),(Root1)}
File    = {(File)}
Root    = {(Root0),(Root1)}
Entry   = {(Entry0),(Entry1),(Entry2),(Entry3)}
Name    = {(Name)}
entries = {(Root1,Entry1),(Root1,Entry2),(Root1,Entry3),(Root1,Entry0)}
object  = {(Entry1,File),(Entry2,File),(Entry0,File),(Entry3,Root0)}
name    = {(Entry0,Name),(Entry1,Name),(Entry2,Name),(Entry3,Name)}
```

Instances as tables

Object
Object
File
Root0
Root1

Dir
Root0
Root1

Root
Root0
Root1

File
File

Name
Name

Entry
Entry0
Entry1
Entry2
Entry3

entries	
Root1	Entry1
Root1	Entry2
Root1	Entry3
Root1	Entry0

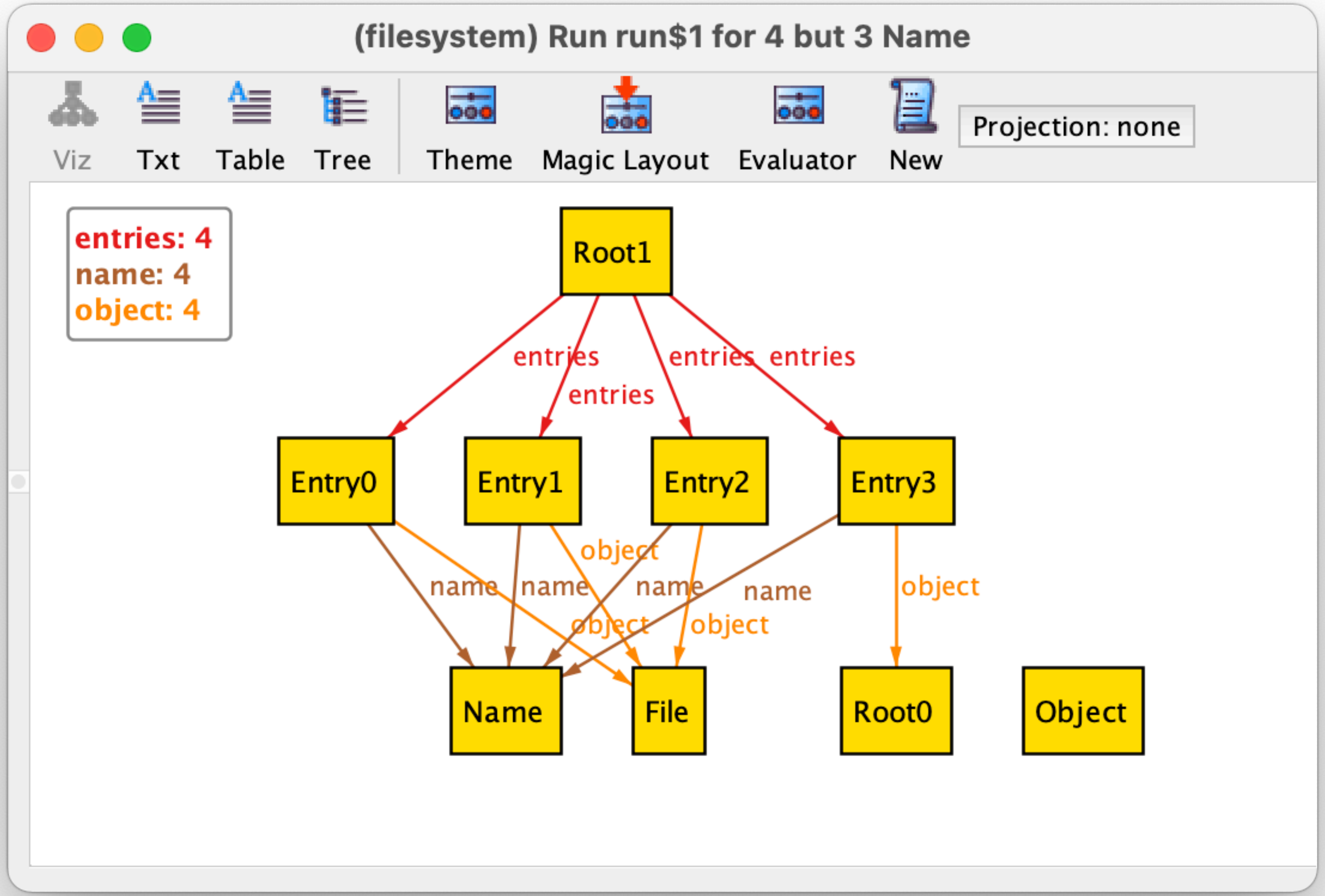
object	
Entry0	File
Entry1	File
Entry2	File
Entry3	Root0

name	
Entry0	Name
Entry1	Name
Entry2	Name
Entry3	Name

Atoms

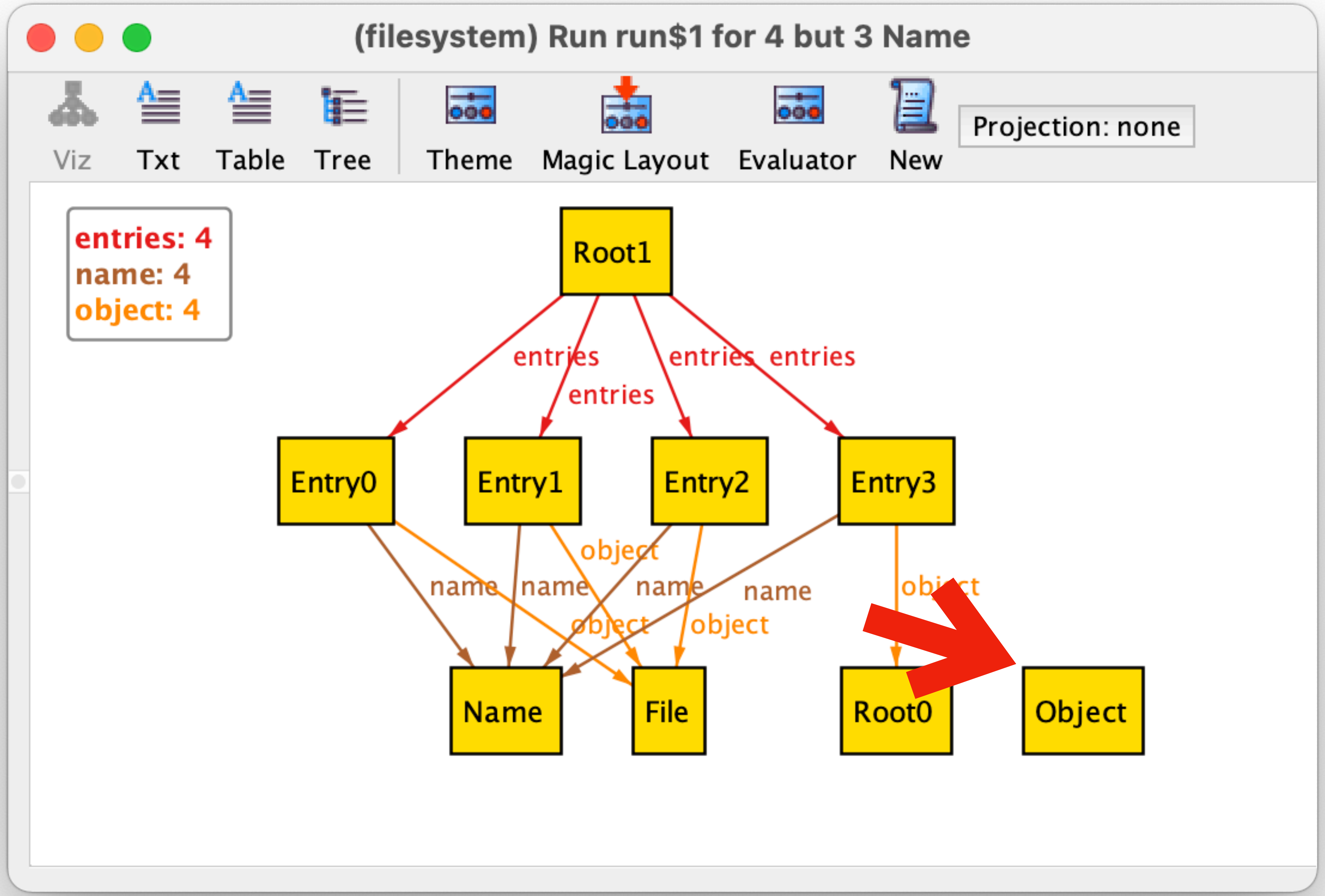
- The universe of discourse contains *atoms*
- Atoms are *uninterpreted* (no semantics)
- Named automatically according to the respective signatures
- Two instances are *isomorphic* (or *symmetric*) if they are equal modulo renaming
- The analysis implements a *symmetry breaking* mechanism to avoid returning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

The constraints



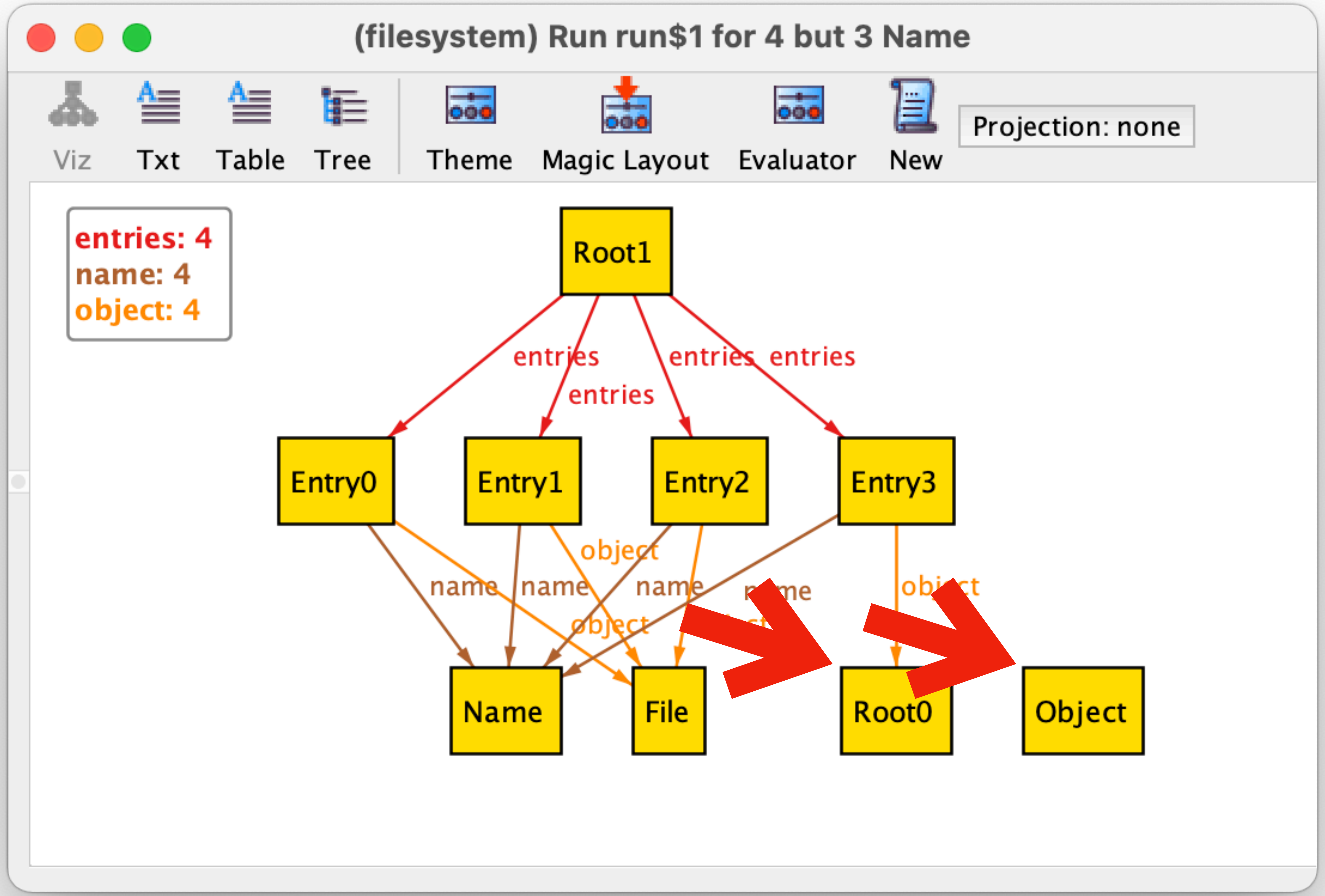
The constraints

- There are no other objects except directories and files



The constraints

- There are no other objects except directories and files
- All objects except the root are contained in at least one entry (at most one for the case of directories)

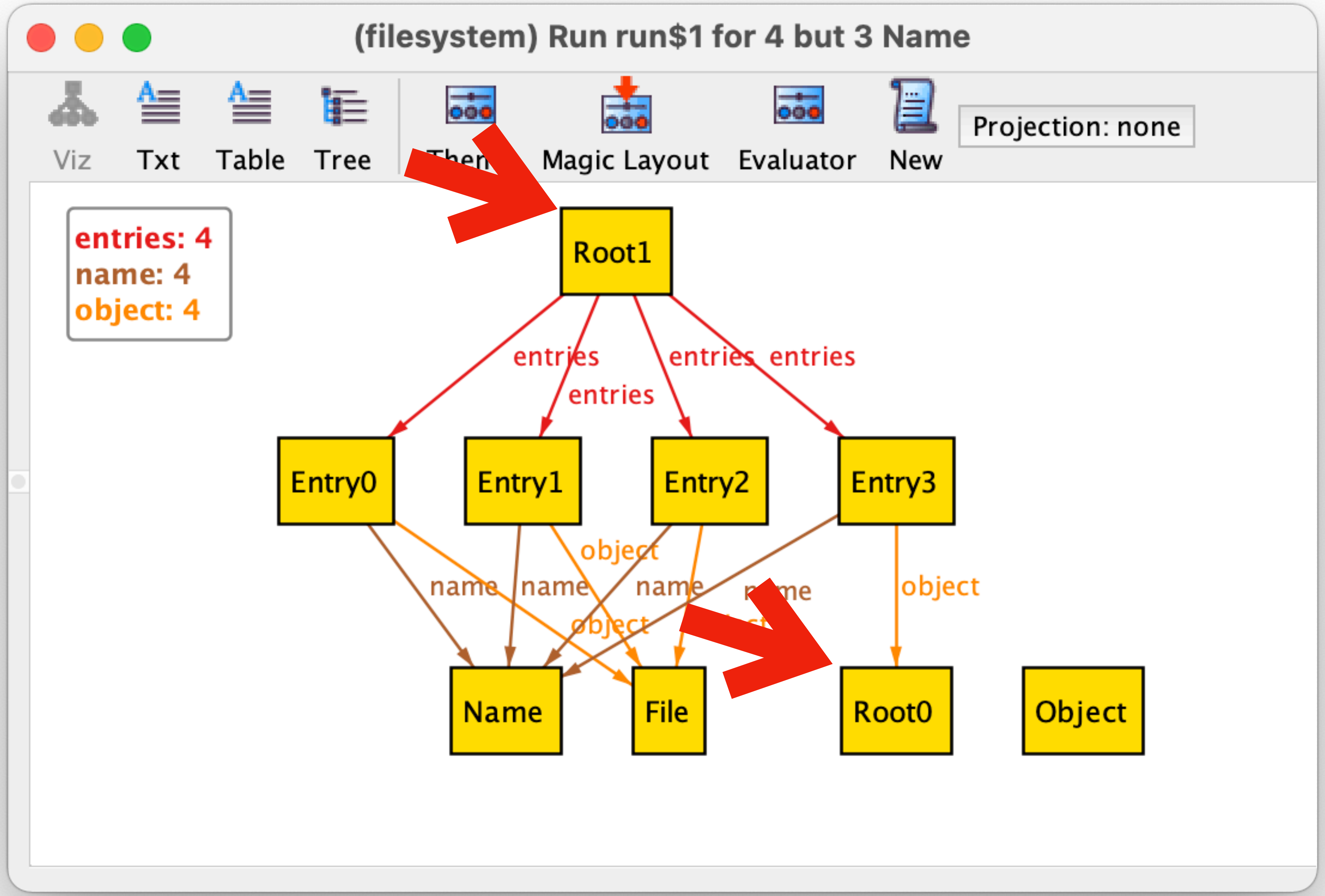


The constraints

- There are no other objects except directories and files

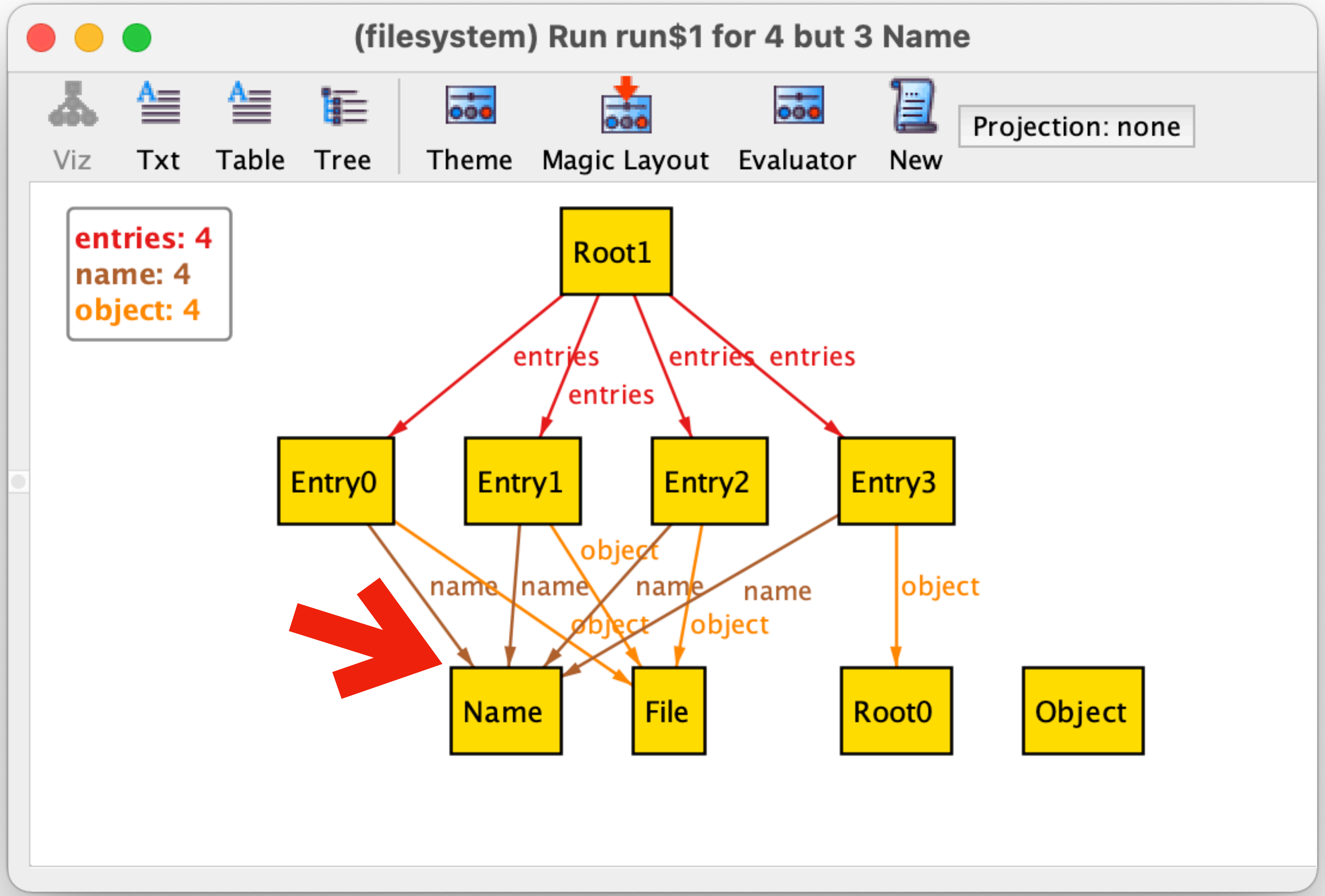
- All objects except the root are contained in at least one entry (at most one for the case of directories)

- There is only one root



The constraints

- There are no other objects except directories and files
- All objects except the root are contained in at least one entry (at most one for the case of directories)
- There is only one root
- Different entries in a directory must have different names



Abstract signatures

- All atoms in an **`abstract`** signature belong to one of its extensions
- The extensions partition the parent signature

```
abstract sig Object {}
sig Dir extends Object {
  entries : set Entry
}
sig File extends Object {}
```

Signature multiplicities

- Multiplicities can also be used in signature declarations
- In particular, a **`one sig`** denotes a constant

```
one sig Root extends Dir {}
```

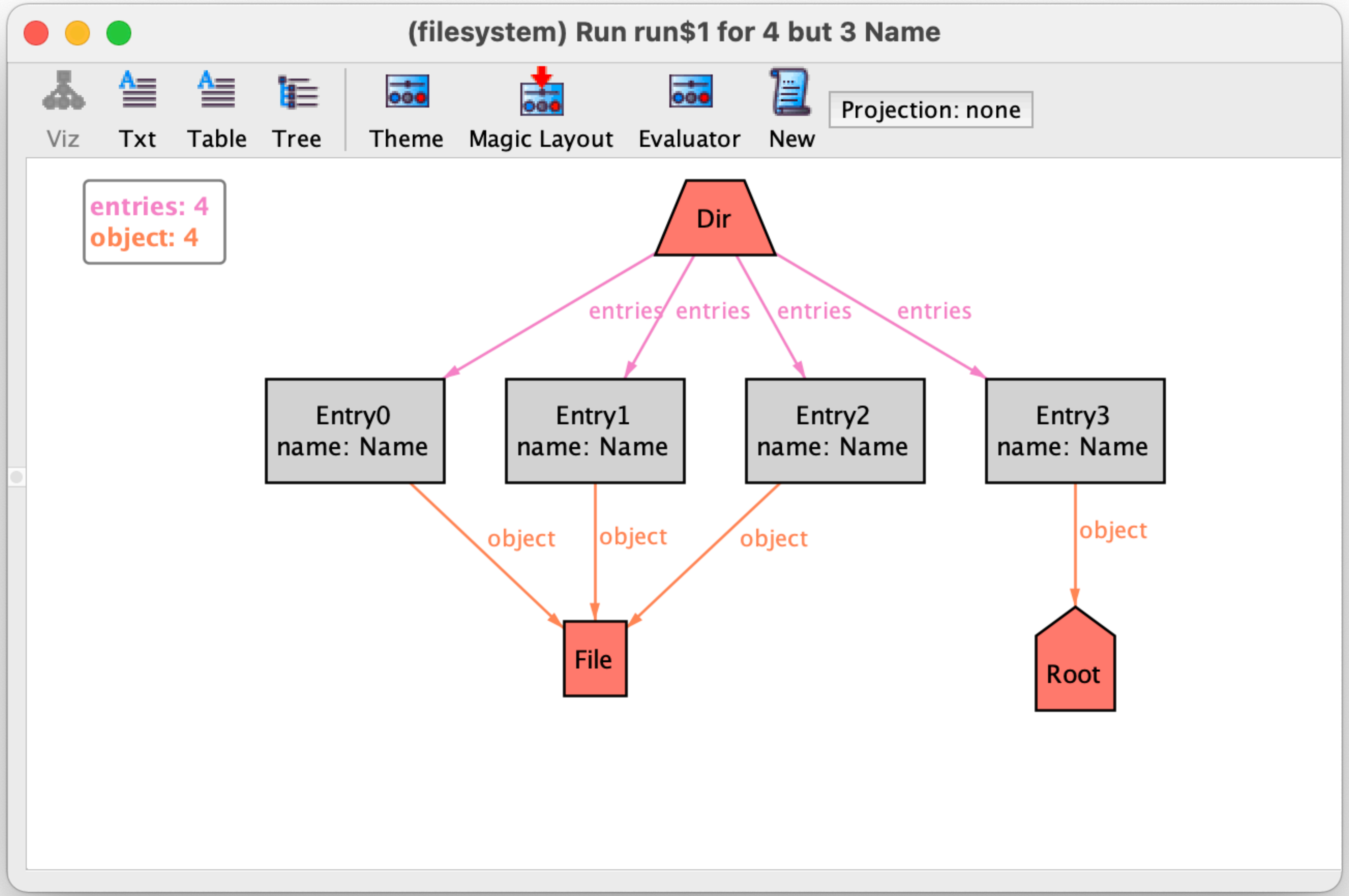
Themes

- The visualiser theme can be customised
- Customisation can ease the understanding and help validate the model
- It is possible to customise colours, shapes, visibility, …

Theme customisation

DEMO

Theme customisation



Relational logic

The constraints in FOL

```
fact {
  // All objects except the root are contained in at least one entry
```
$\forall o \cdot \texttt{Object}(o) \land o \neq \texttt{Root} \rightarrow \exists e \cdot \texttt{object}(e, d)$

$\forall e \,.\, \neg\texttt{object}(e, \texttt{Root})$

```
  // All directories are contained in at most one entry
```
$\forall d, e_1, e_2 \cdot \texttt{Dir}(d) \land \texttt{object}(e_1, d) \land \texttt{object}(e_2, d) \rightarrow e_1 = e_2$

```
  // Different entries in a directory must have different names
```
$\forall d, n, e_1, e_2 \cdot \texttt{entries}(d, e_1) \land \texttt{entries}(d, e_2) \land \texttt{name}(e_1, n) \land \texttt{name}(e_2, n) \rightarrow e_1 = e_2$

```
}
```

The constraints in FOL

```
fact {
  // All objects except the root are contained in at least one entry
```
$\forall o \cdot (o) \in \texttt{Object} \wedge o \neq \texttt{Root} \rightarrow \exists e \cdot (e, d) \in \texttt{object}$

$\forall e \,.\, (e, \texttt{Root}) \notin \texttt{object}$

```
  // All directories are contained in at most one entry
```
$\forall d, e_1, e_2 \cdot (d) \in \texttt{Dir} \wedge (e_1, d) \in \texttt{object} \wedge (e_2, d) \in \texttt{object} \rightarrow e_1 = e_2$

```
  // Different entries in a directory must have different names
```
$\forall d, n, e_1, e_2 \cdot (d, e_1) \in \texttt{entries} \wedge (d, e_2) \in \texttt{entries} \wedge (e_1, n) \in \texttt{name} \wedge (e_2, n) \in \texttt{name} \rightarrow e_1 = e_2$

```
}
```

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	$\phi \leftrightarrow \psi$

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	$\phi \leftrightarrow \psi$

Quantifiers

all x : **univ** | ϕ	$\forall x \cdot \phi$
all x : A | ϕ	**all** x : **univ** | x **in** A => ϕ
some x : **univ** | ϕ	$\exists x \cdot \phi$
some x : A | ϕ	**some** x : **univ** | x **in** A && ϕ

Atomic formulas

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

The constraints in Alloy

```
fact {
  // All objects except the root are contained in at least one entry
  all o : univ | o in Object and o != Root implies some e : univ | e->o in object
  all o : univ | o->Root not in object

  // All directories are contained in at most one entry
  all d,e1,e2 : univ | d in Dir and e1->d in object and e2->d in object implies e1 = e2

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

Relational logic

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

Atomic formulas

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

Atomic formulas

```
// Cardinality checks
```

`some`	R	$\|R\| > 0$	$\exists x_1, \ldots, x_n \cdot (x_1, \ldots, x_n) \in R$
`no`	R	$\|R\| = 0$	$\forall x_1, \ldots, x_n \cdot (x_1, \ldots, x_n) \notin R$
`lone`	R	$\|R\| < 2$	
`one`	R	$\|R\| = 1$	

Set operators

```
// Union
```

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

```
// Intersection
```

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

```
// Difference
```

$R - S$ $\quad R \setminus S \quad (x_1, \ldots, x_n) \in (R - S) \leftrightarrow (x_1, \ldots, x_n) \in R \wedge (x_1, \ldots, x_n) \notin S$

Relational constants

```
// Universe
```

`univ` $\top$ $\forall x \cdot (x) \in \texttt{\textbf{univ}}$

```
// Empty set
```

`none` $\varnothing$ $\forall x \cdot (x) \notin \texttt{\textbf{none}}$

```
// Identity
```

`iden` id $\forall x_1, x_2 \cdot (x_1, x_2) \in \texttt{\textbf{iden}} \leftrightarrow x_1 = x_2$

Relational operators

```
// Cartesian product
```
R `->` S $\quad R \times S \quad (x_1, \ldots, x_n, y_1, \ldots, y_m) \in (R$ `->` $S) \leftrightarrow (x_1, \ldots, x_n) \in R \wedge (y_1, \ldots, y_m) \in S$

```
// Transpose or converse
```
$\sim R \qquad R^{\circ} \qquad (x_1, x_2) \in (\sim R) \leftrightarrow (x_2, x_1) \in R$

```
// Range restriction
```
R `:>` $A \qquad (x_1, \ldots, x_n) \in (R$ `:>` $A) \leftrightarrow (x_1, \ldots, x_n) \in R \wedge (x_n) \in A$

```
// Domain restriction
```
A `<:` $R \qquad (x_1, \ldots, x_n) \in (A$ `<:` $R) \leftrightarrow (x_1, \ldots, x_n) \in R \wedge (x_1) \in A$

Inclusion vs subset

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

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

~~$\forall x\,.\,(x) \in$ `Dir` $\rightarrow (x) \in$ `Object`~~

$\forall x\,.\,\{(x)\} \subseteq$ `Dir` $\rightarrow \{(x)\} \subseteq$ `Object`

Inclusion vs subset

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

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

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

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

Composition

$$R \;\fatsemi\; S$$

$$S \circ R$$

$$(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m) \in (R \;\fatsemi\; S)$$

$$\leftrightarrow$$

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

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

`entries . name`

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

entries . name	
Root	Name0

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

entries . name	
Root	Name0
Root	Name1

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

entries . name	
Root	Name0
Root	Name1
Dir	Name1

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

entries . name	
Root	Name0
Root	Name1
Dir	Name1

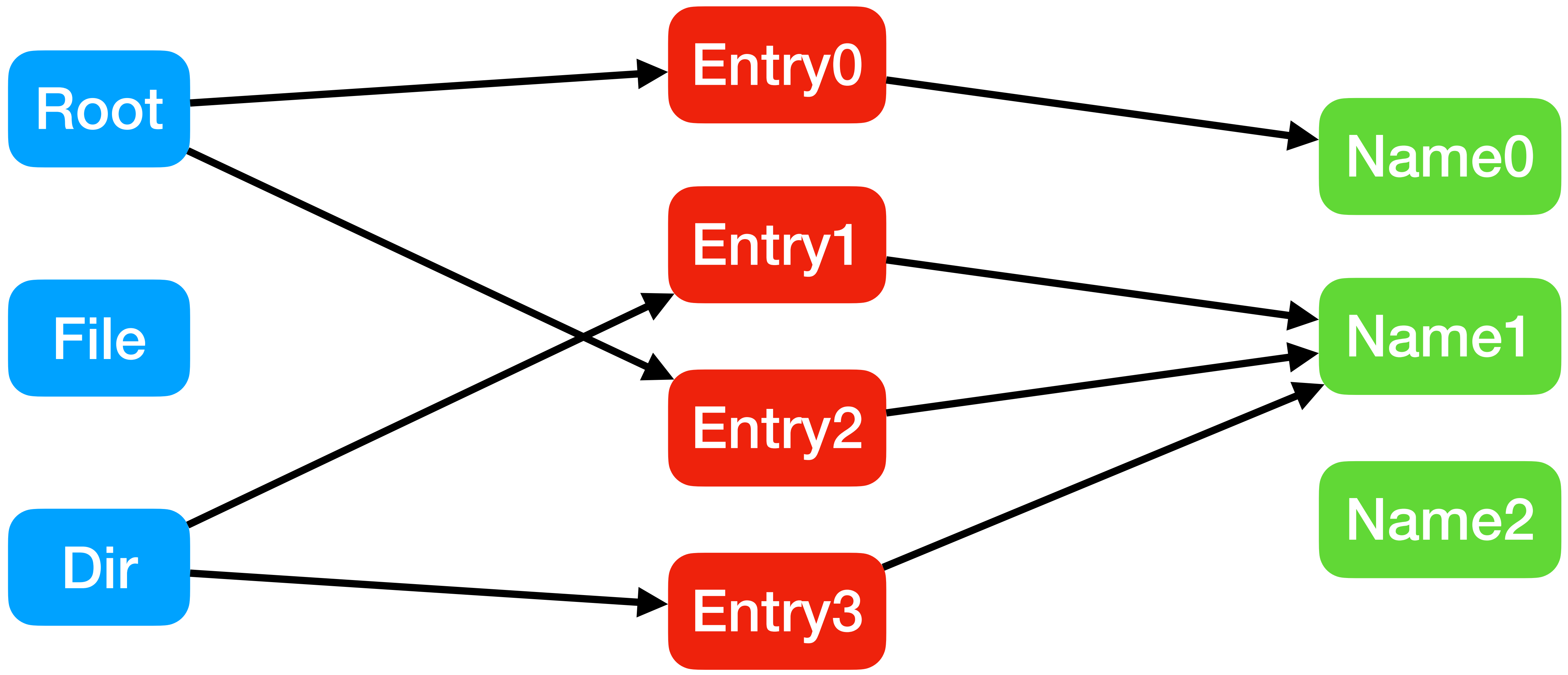
Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

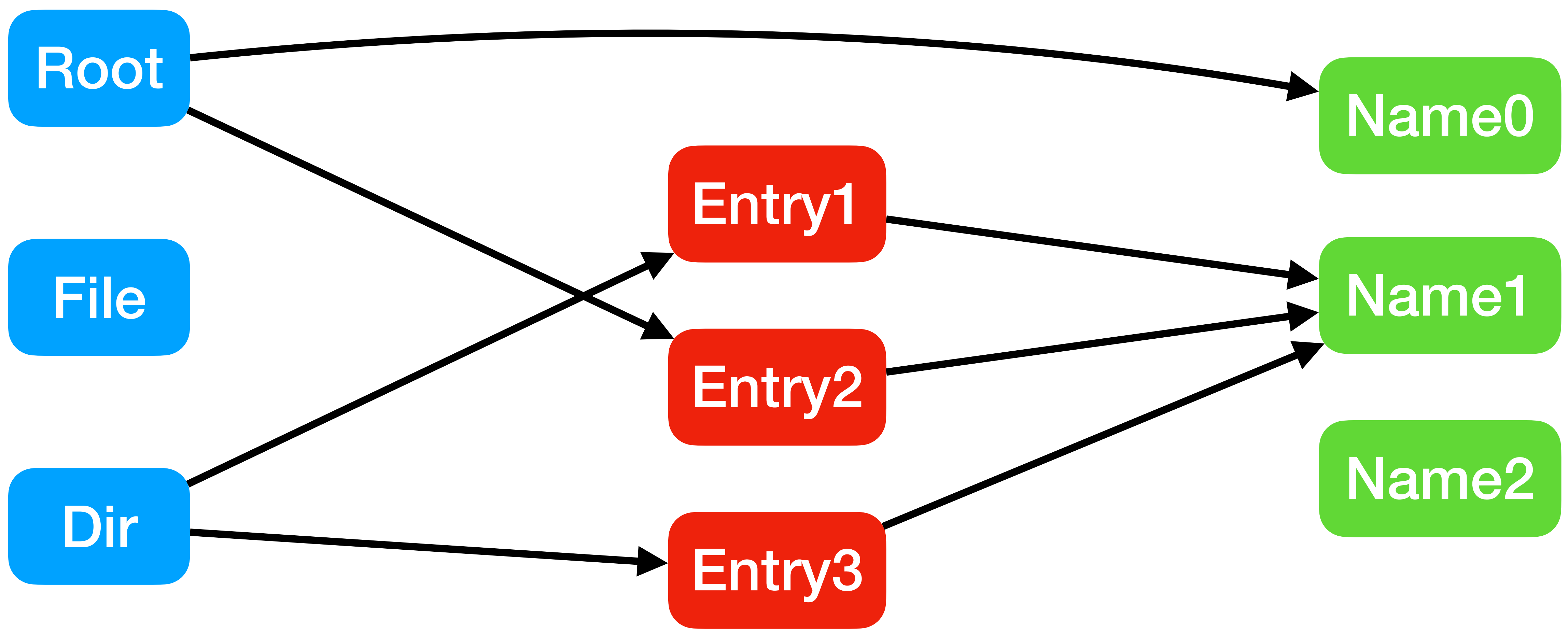
name	
Entry0	Name0
Entry1	Name1
Entry2	Name1
Entry3	Name1

entries . name	
Root	Name0
Root	Name1
Dir	Name1

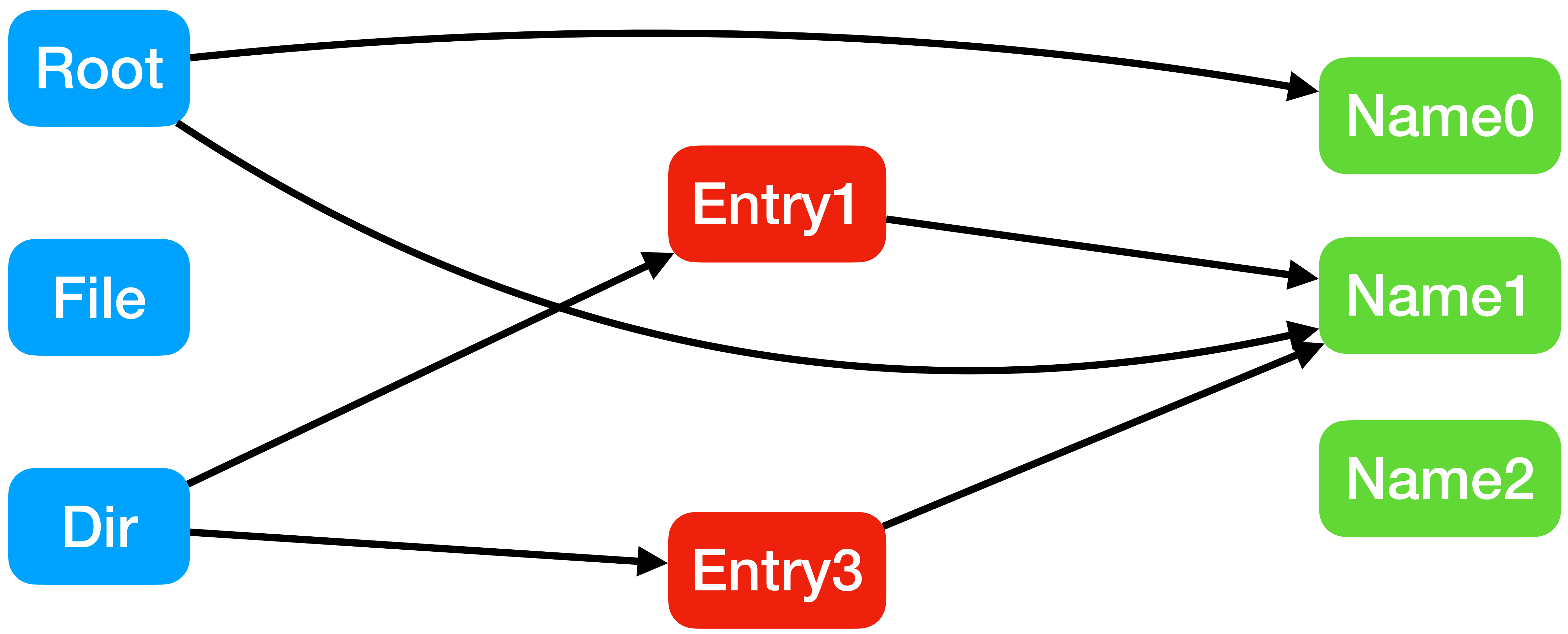
Composition



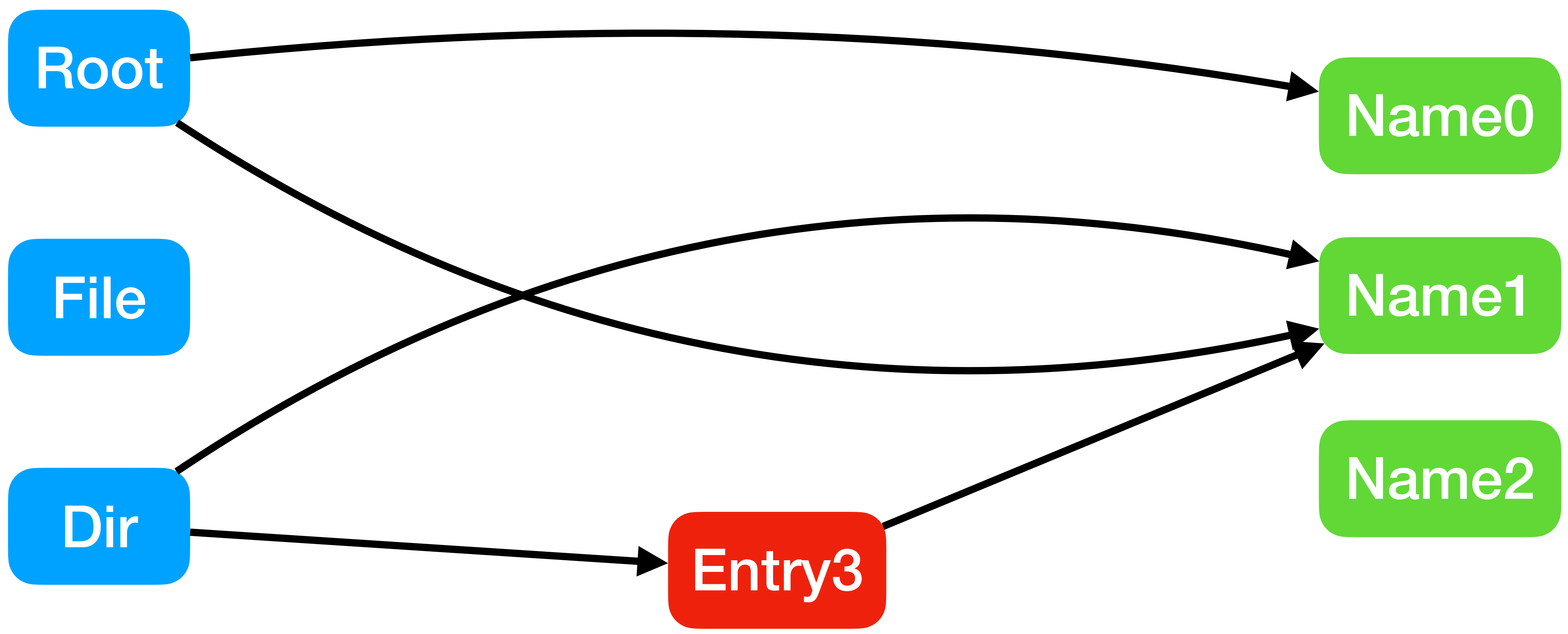
Composition



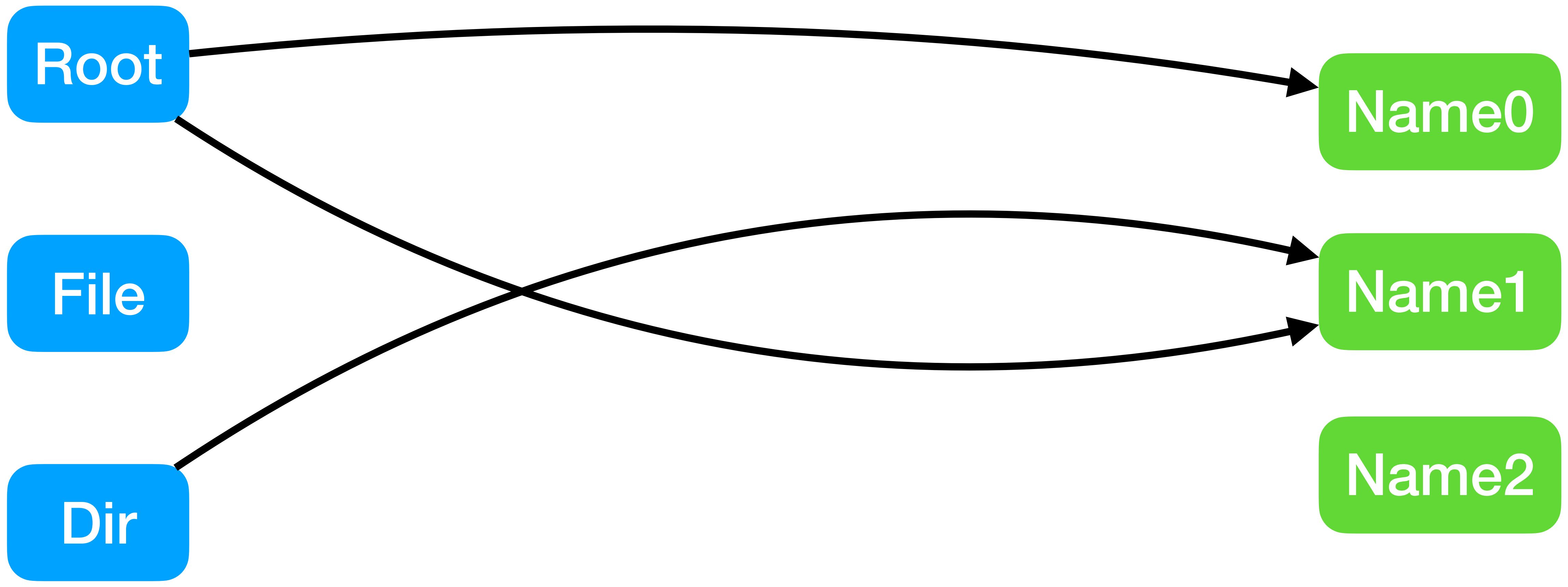
Composition



Composition



Composition



Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

`entries . Entry`

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

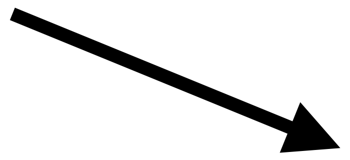
→

Entry
Entry0
Entry1
Entry2
Entry3

entries . Entry
Root

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3



Entry
Entry0
Entry1
Entry2
Entry3

entries . Entry
Root

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

→

Entry
Entry0
Entry1
Entry2
Entry3

entries . Entry
Root
Dir

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

entries . Entry
Root
Dir

Composition

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Entry
Entry0
Entry1
Entry2
Entry3

entries . Entry
Root
Dir

Composition

Root
Root

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Composition

Root
Root

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Root . entries

Composition

Root
Root

→

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Root . entries
Entry0

Composition

Root
Root

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Root . entries
Entry0
Entry2

Composition

Root
Root

entries	
Root	Entry0
Root	Entry2
Dir	Entry1
Dir	Entry3

Root . entries
Entry0
Entry2

From FOL to RL

From FOL to RL

```
all o : univ | o->Root not in object
```

From FOL to RL

```
all o : univ | o->Root not in object

all o : univ | o not in object.Root
```

From FOL to RL

```
all o : univ | o->Root not in object

all o : univ | o not in object.Root

no object.Root
```

From FOL to RL

From FOL to RL

```
all o : univ | o in Object and o != Root implies some e : univ | e->o in object
```

From FOL to RL

```
all o : univ | o in Object and o != Root implies some e : univ | e->o in object

all o : univ | o in Object and o != Root implies some e : univ | e in object.o
```

From FOL to RL

```
all o : univ | o in Object and o != Root implies some e : univ | e->o in object

all o : univ | o in Object and o != Root implies some e : univ | e in object.o

all o : univ | o in Object and o != Root implies some object.o
```

From FOL to RL

```
all o : univ | o in Object and o != Root implies some e : univ | e->o in object

all o : univ | o in Object and o != Root implies some e : univ | e in object.o

all o : univ | o in Object and o != Root implies some object.o

all o : univ | o in Object and o not in Root implies some object.o
```

From FOL to RL

```
all o : univ | o in Object and o != Root implies some e : univ | e->o in object

all o : univ | o in Object and o != Root implies some e : univ | e in object.o

all o : univ | o in Object and o != Root implies some object.o

all o : univ | o in Object and o not in Root implies some object.o

all o : univ | o in Object-Root implies some object.o
```

From FOL to RL

```
all o : univ | o in Object and o != Root implies some e : univ | e->o in object

all o : univ | o in Object and o != Root implies some e : univ | e in object.o

all o : univ | o in Object and o != Root implies some object.o

all o : univ | o in Object and o not in Root implies some object.o

all o : univ | o in Object-Root implies some object.o

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

From FOL to RL

From FOL to RL

```
all d,e1,e2 : univ | d in Dir and e1->d in object and e2->d in object implies e1 = e2
```

From FOL to RL

```
all d,e1,e2 : univ | d in Dir and e1->d in object and e2->d in object implies e1 = e2

all d : univ | d in Dir implies all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2
```

From FOL to RL

```
all d,e1,e2 : univ | d in Dir and e1->d in object and e2->d in object implies e1 = e2

all d : univ | d in Dir implies all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2

all d : Dir | all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2
```

From FOL to RL

```
all d,e1,e2 : univ | d in Dir and e1->d in object and e2->d in object implies e1 = e2

all d : univ | d in Dir implies all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2

all d : Dir | all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2

all d : Dir | all e1,e2 : univ | e1 in object.d and e2 in object.d implies e1 = e2
```

From FOL to RL

```
all d,e1,e2 : univ | d in Dir and e1->d in object and e2->d in object implies e1 = e2

all d : univ | d in Dir implies all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2

all d : Dir | all e1,e2 : univ | e1->d in object and e2->d in object implies e1 = e2

all d : Dir | all e1,e2 : univ | e1 in object.d and e2 in object.d implies e1 = e2

all d : Dir | lone object.d
```

From FOL to RL

From FOL to RL

```
all d,n,e1,e2 : univ | d->e1 in entries and d->e2 in entries and e1->n in name and e2->n in name implies e1 = e2
```

From FOL to RL

```
all d,n,e1,e2 : univ | d->e1 in entries and d->e2 in entries and e1->n in name and e2->n in name implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e2 in entries.d and e1 in name.n and e2 in name.n implies e1 = e2
```

From FOL to RL

```
all d,n,e1,e2 : univ | d->e1 in entries and d->e2 in entries and e1->n in name and e2->n in name implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e2 in entries.d and e1 in name.n and e2 in name.n implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e1 in name.n and e2 in entries.d and e2 in name.n implies e1 = e2
```

From FOL to RL

```
all d,n,e1,e2 : univ | d->e1 in entries and d->e2 in entries and e1->n in name and e2->n in name implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e2 in entries.d and e1 in name.n and e2 in name.n implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e1 in name.n and e2 in entries.d and e2 in name.n implies e1 = e2

all d,n,e1,e2 : univ | e1 in (entries.d & name.n) and e2 in (entries.d & name.n) implies e1 = e2
```

From FOL to RL

```
all d,n,e1,e2 : univ | d->e1 in entries and d->e2 in entries and e1->n in name and e2->n in name implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e2 in entries.d and e1 in name.n and e2 in name.n implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e1 in name.n and e2 in entries.d and e2 in name.n implies e1 = e2

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

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

From FOL to RL

```
all d,n,e1,e2 : univ | d->e1 in entries and d->e2 in entries and e1->n in name and e2->n in name implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e2 in entries.d and e1 in name.n and e2 in name.n implies e1 = e2

all d,n,e1,e2 : univ | e1 in entries.d and e1 in name.n and e2 in entries.d and e2 in name.n implies e1 = e2

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

all d,n : univ | lone (entries.d & name.n)

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

The constraints in Alloy

```
fact {
  // All objects except the root are contained in at least one entry
  all o : Object - Root | some object.o
  no object.Root

  // All directories are contained in at most one entry
  all d : Dir | lone object.d

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

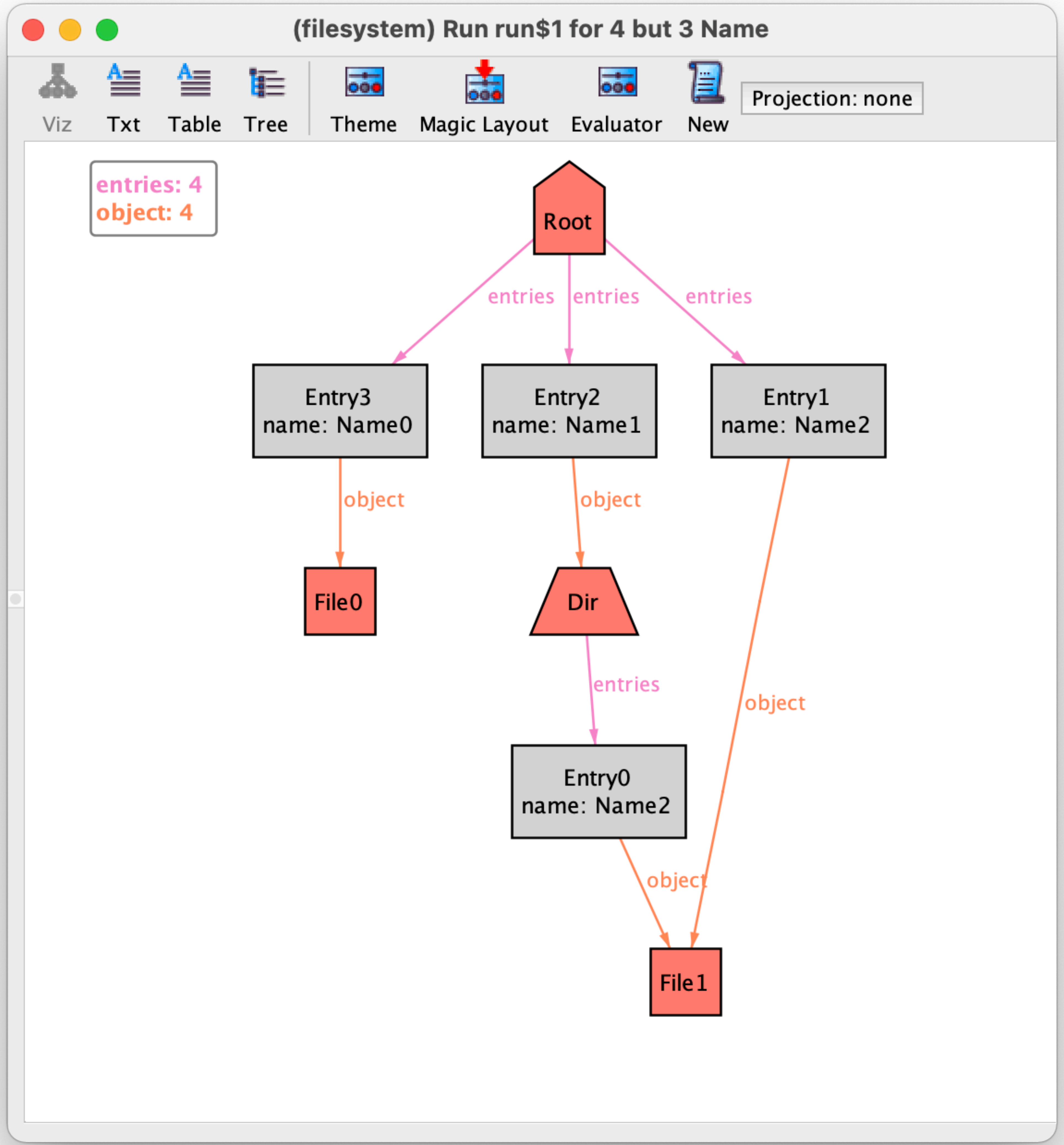
```
// First order style
all x,y : Entry, n : Name | x->n in name and y->n in name implies x=y

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

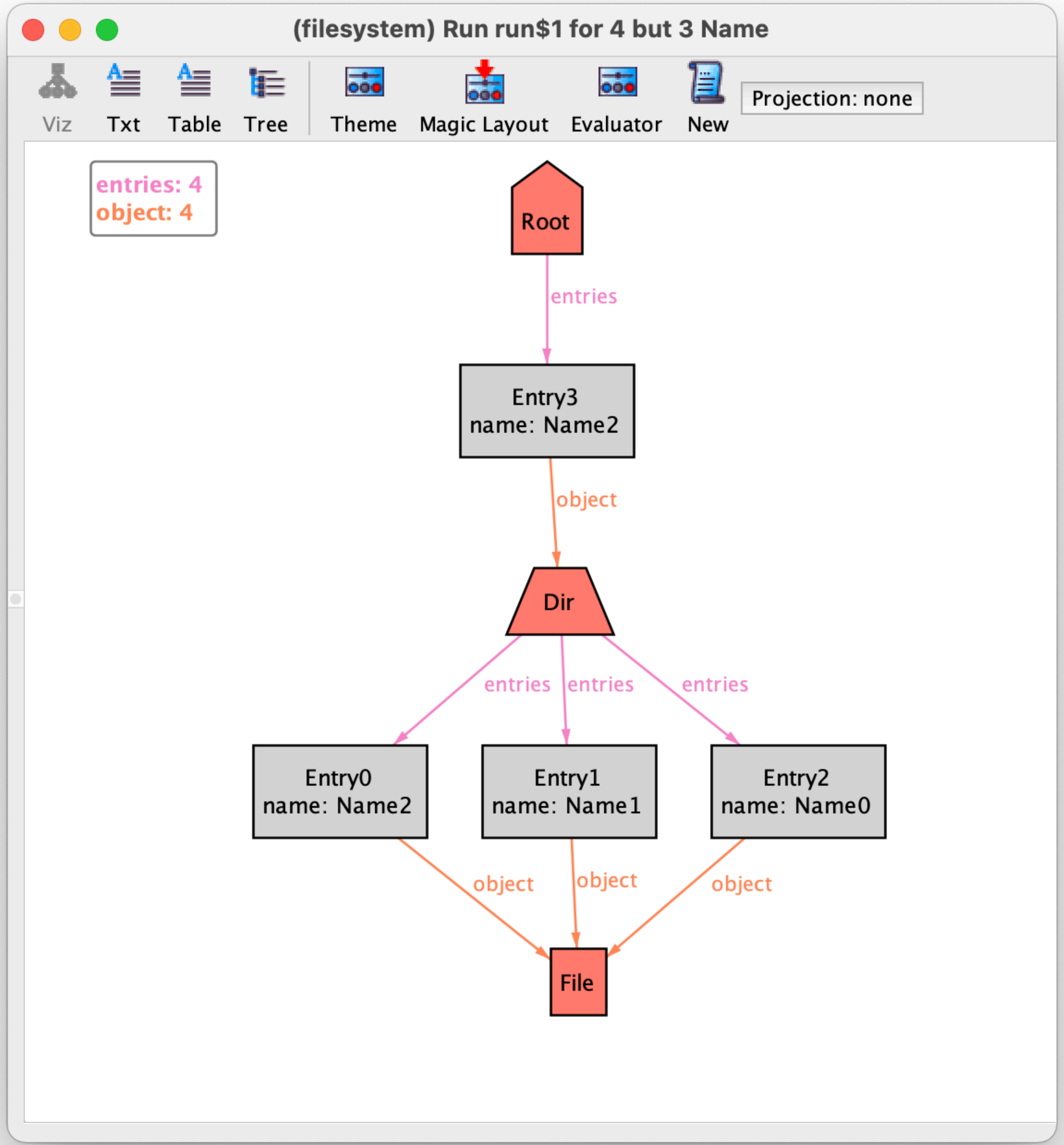
A question of style

```
// 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



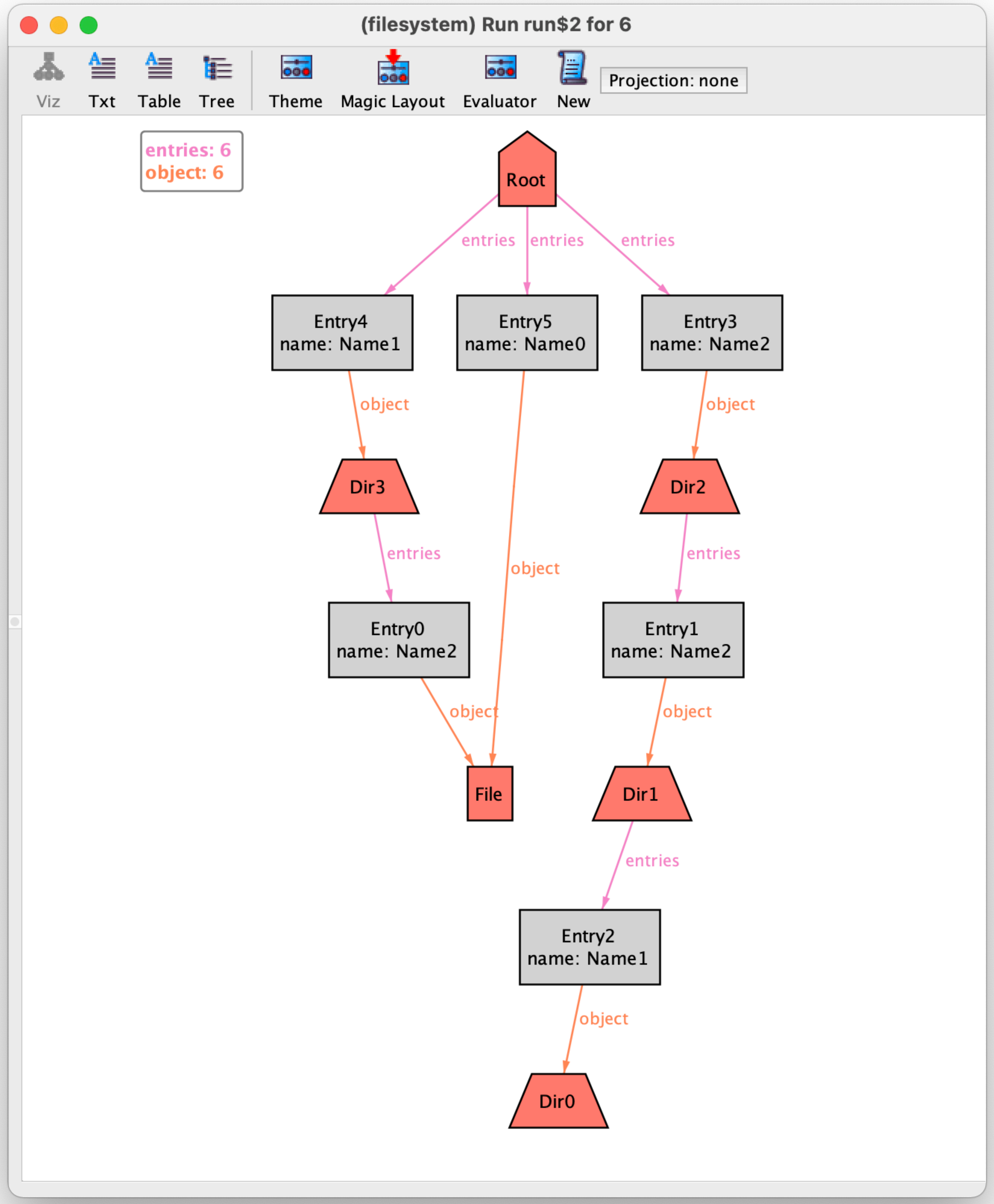


Assertions

- Assertions are named constraints to be checked

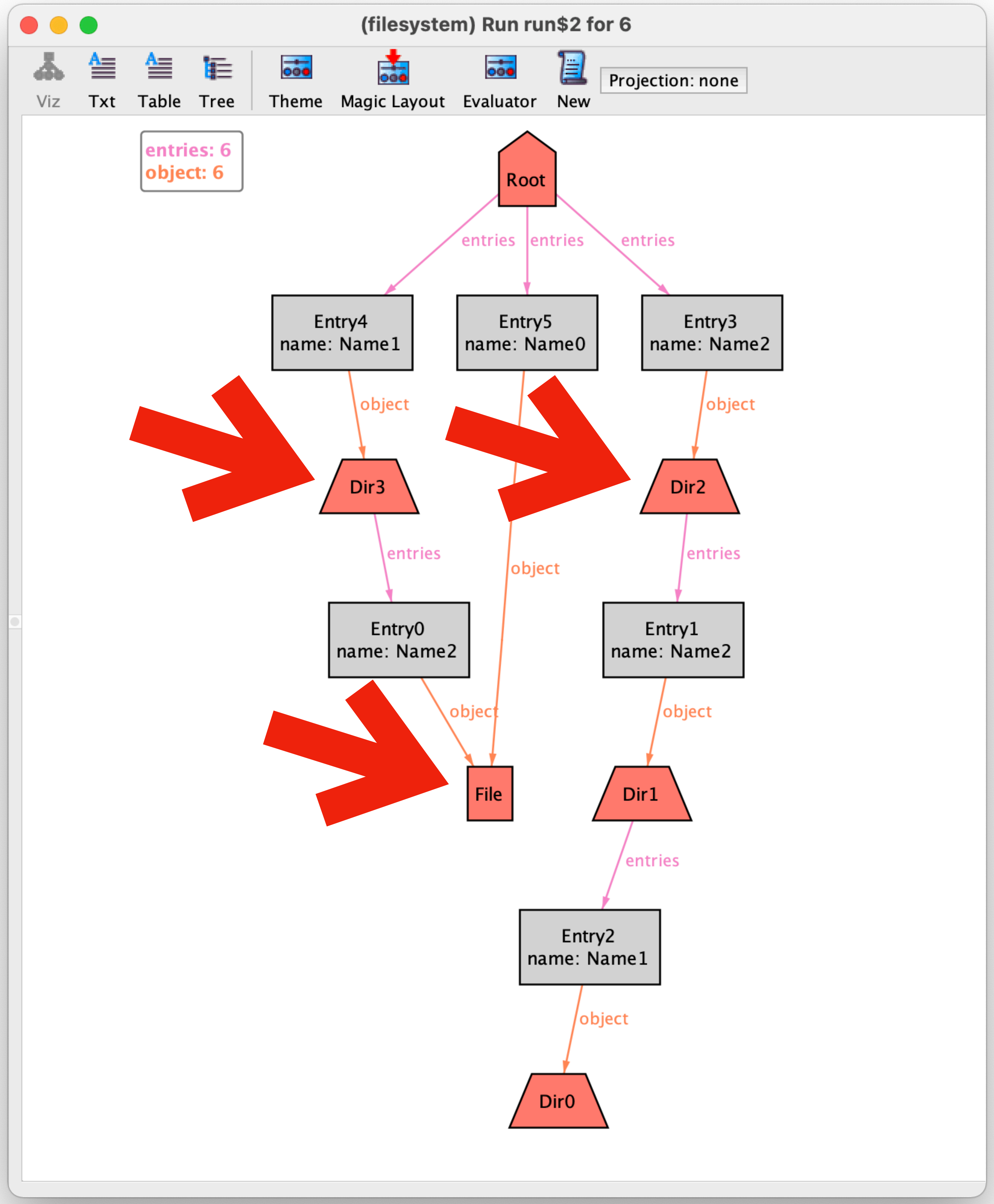
```
assert NoPartitions {
  // All objects are reachable from the root
  ???
}

check NoPartitions
```

Reachable objects



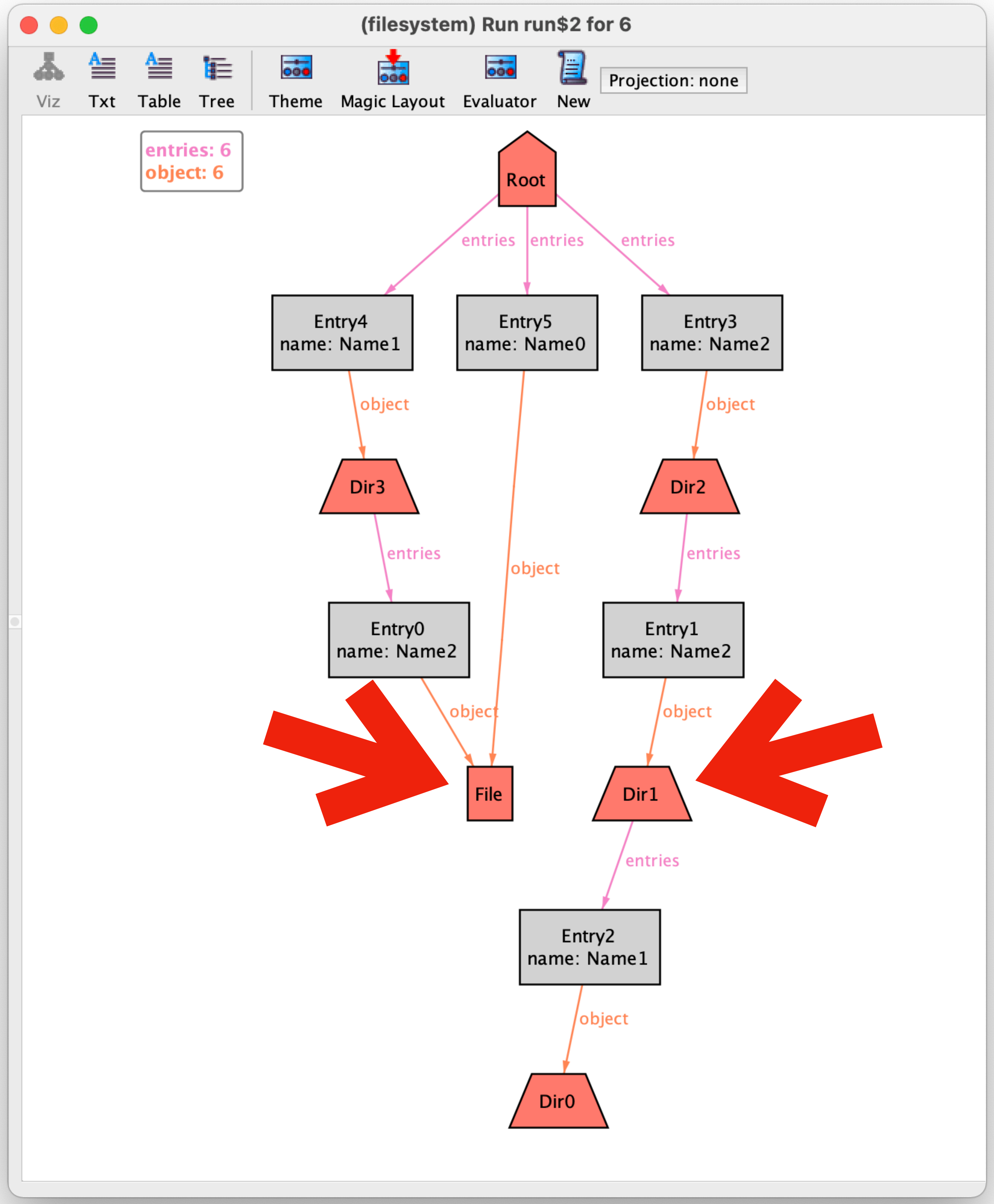
Reachable objects

`Root.entries.object`



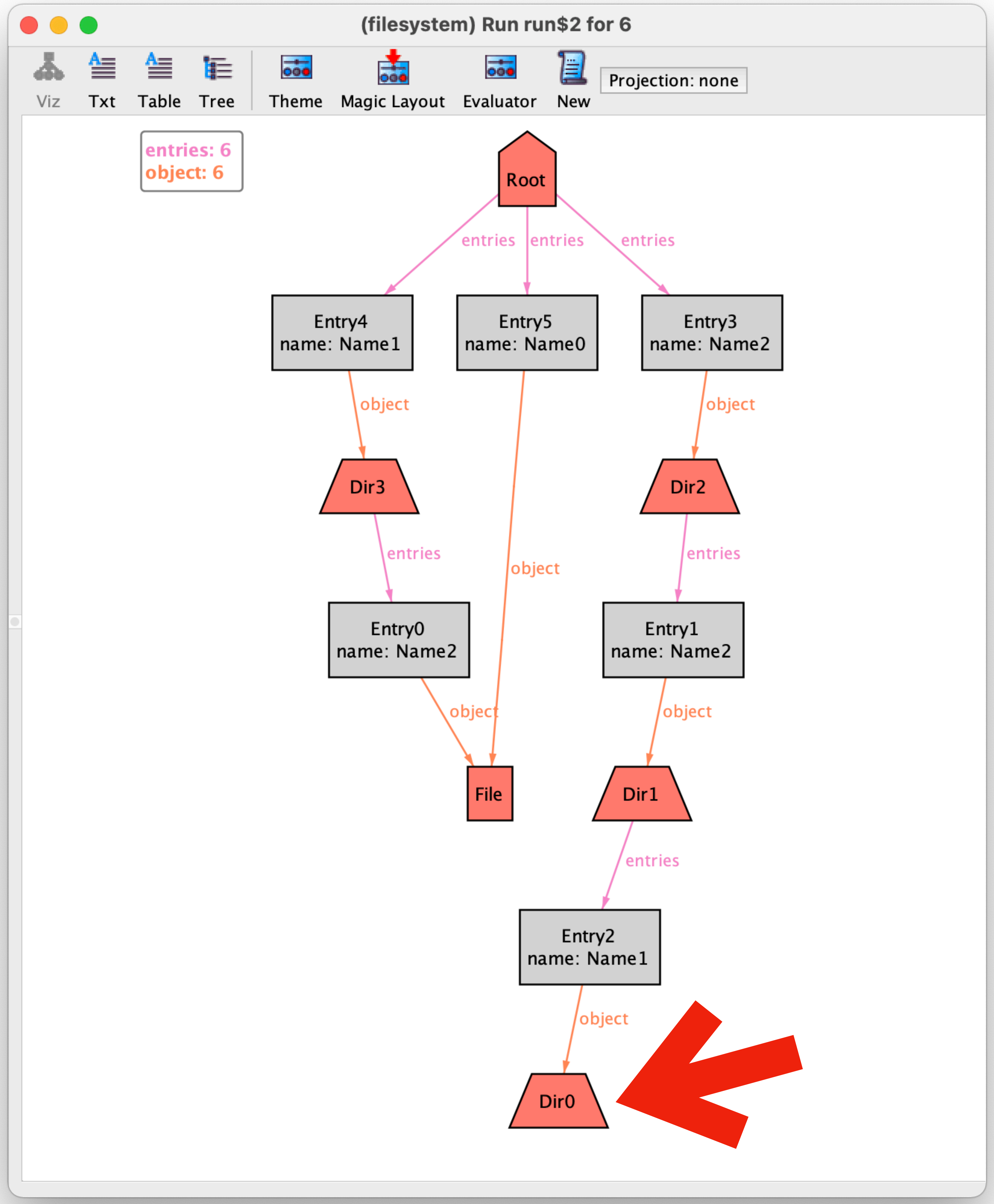
Reachable objects

`Root.entries.object.entries.object`



Reachable objects

`Root.entries.object.entries.object.entries.object`



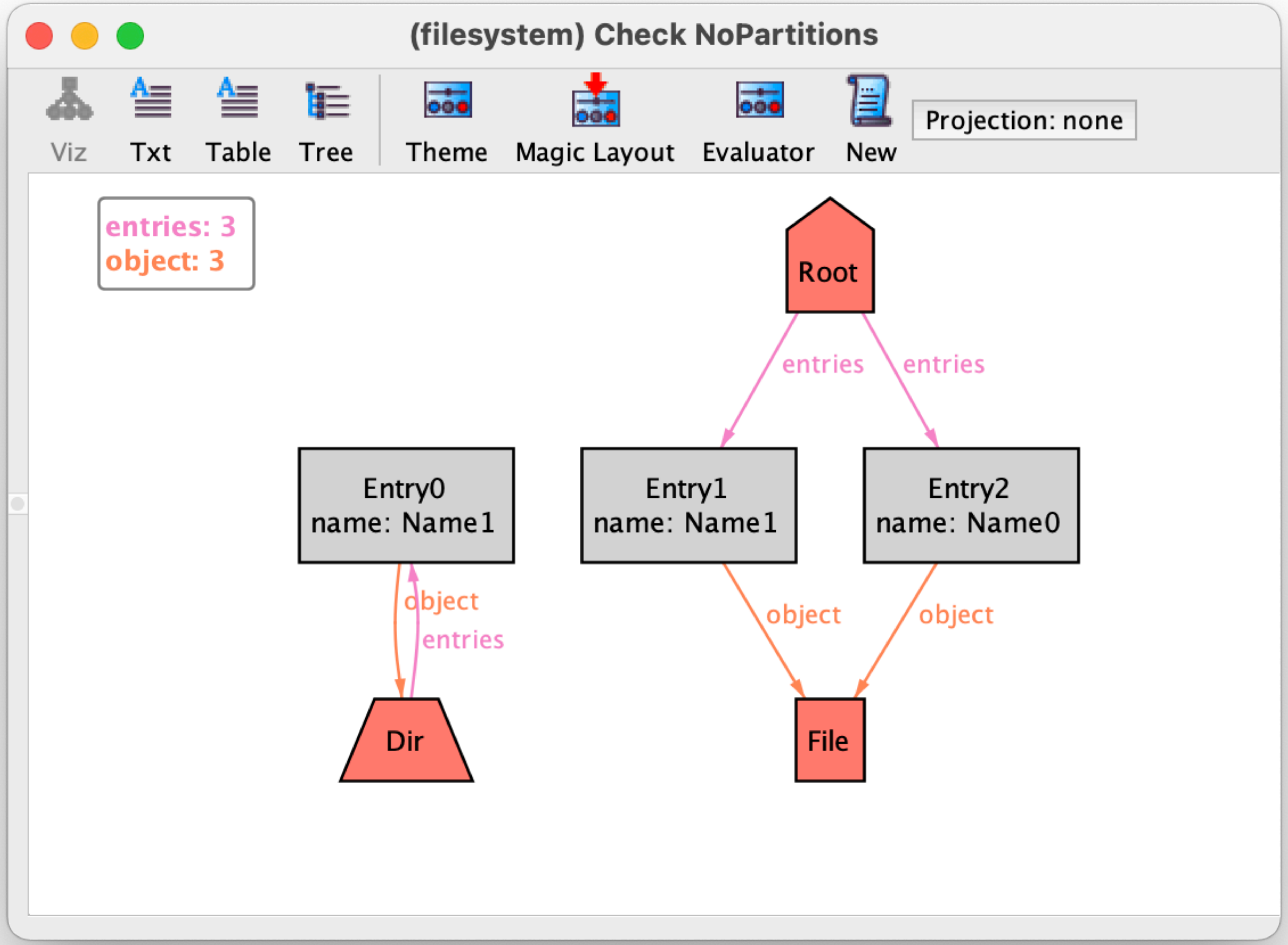
Closures

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

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

The desired assertion

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

check NoPartitions
```

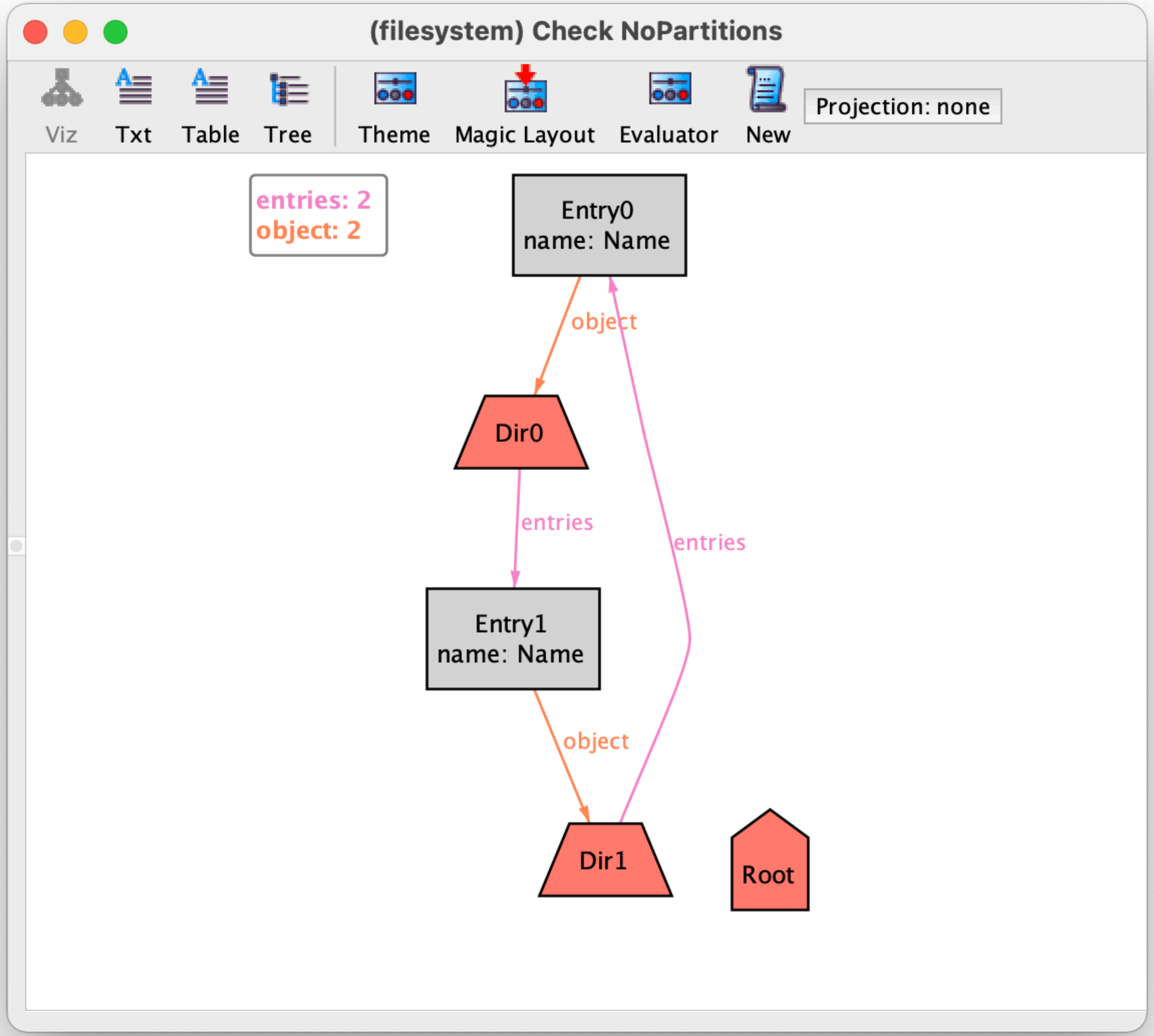
A counter-example



The missing constraint

```
fact {
  // All objects except the root are contained in at least one entry
  all o : Object - Root | some object.o
  no object.Root

  // All directories are contained in at most one entry
  all d : Dir | lone object.d

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

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

Another counter-example



The missing constraint

```
fact {
  // All objects except the root are contained in at least one entry
  all o : Object - Root | some object.o
  no object.Root

  // All directories are contained in at most one entry
  all d : Dir | lone object.d

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

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

```
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

- Verify that specific scenarios are possible

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