

Structural design with Alloy

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

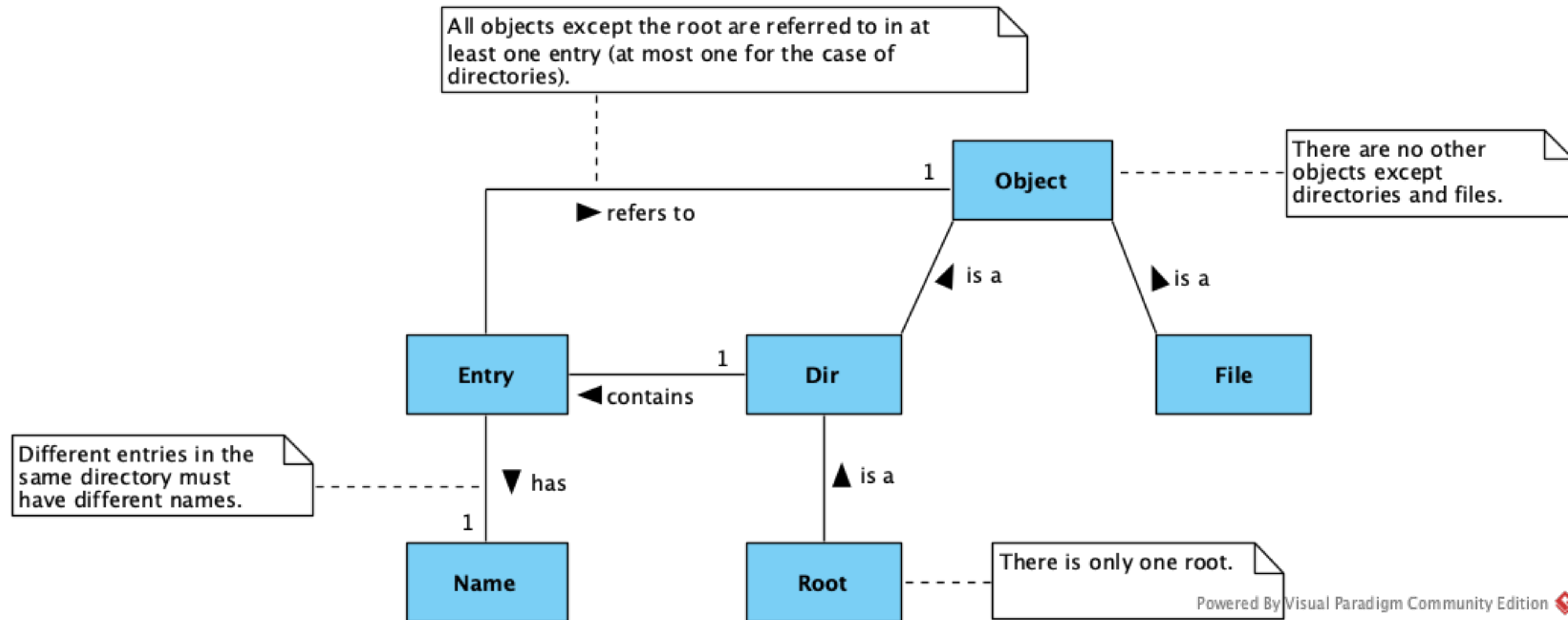
Software structures

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

Structural design

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

Domain modeling *à la* UML



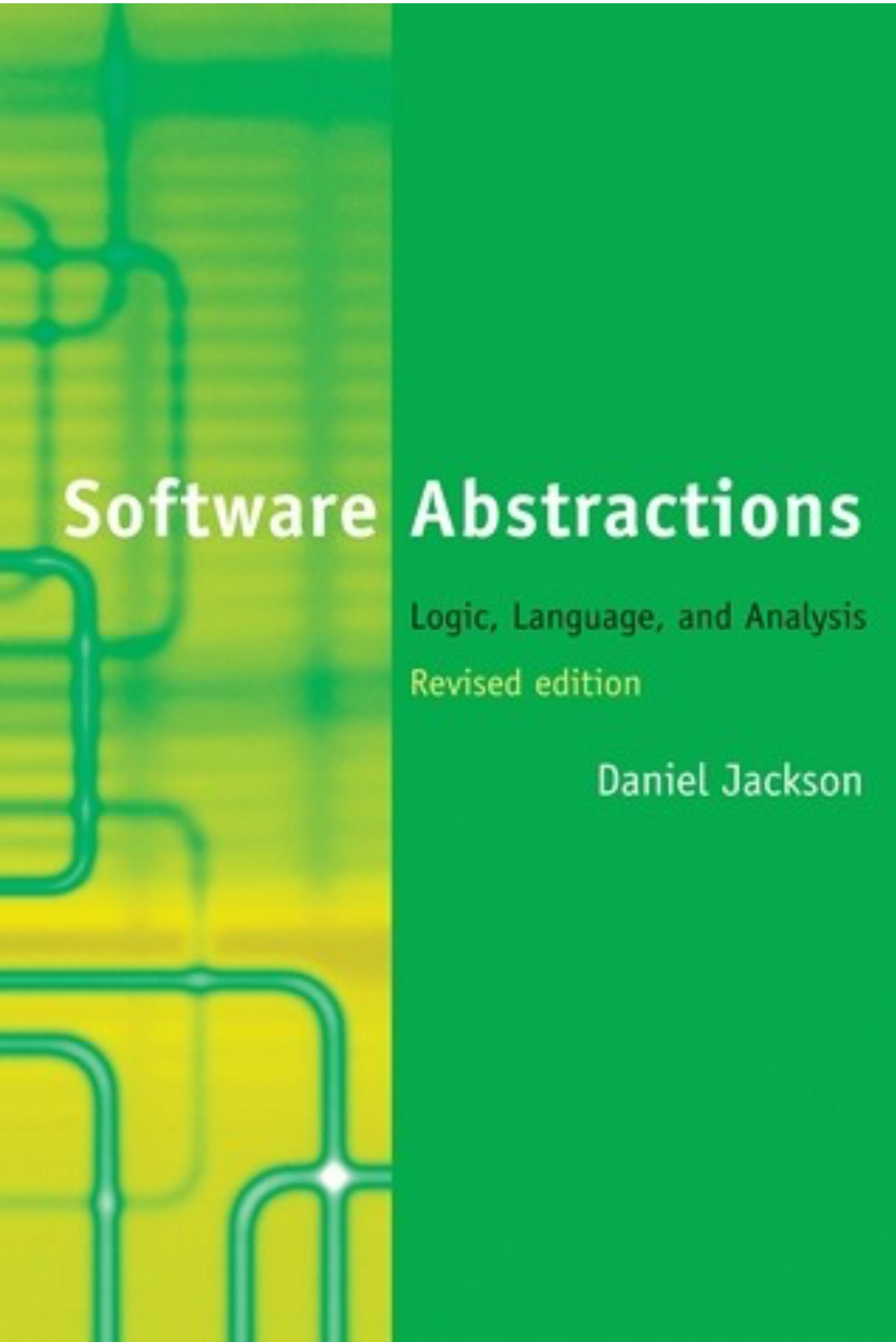
Requirement elicitation

- Are the requirements consistent?
- Any forgotten or redundant requirements?
- Do the requirements entail all the expected properties?

Alloy

Software design with Alloy

- Alloy is a formal modeling language
- Based on relational logic, an extension of first-order logic
- Models can be automatically analyzed
- Tailored for abstraction - everything is a relation!



Software Abstractions

Logic, Language, and Analysis

Revised edition

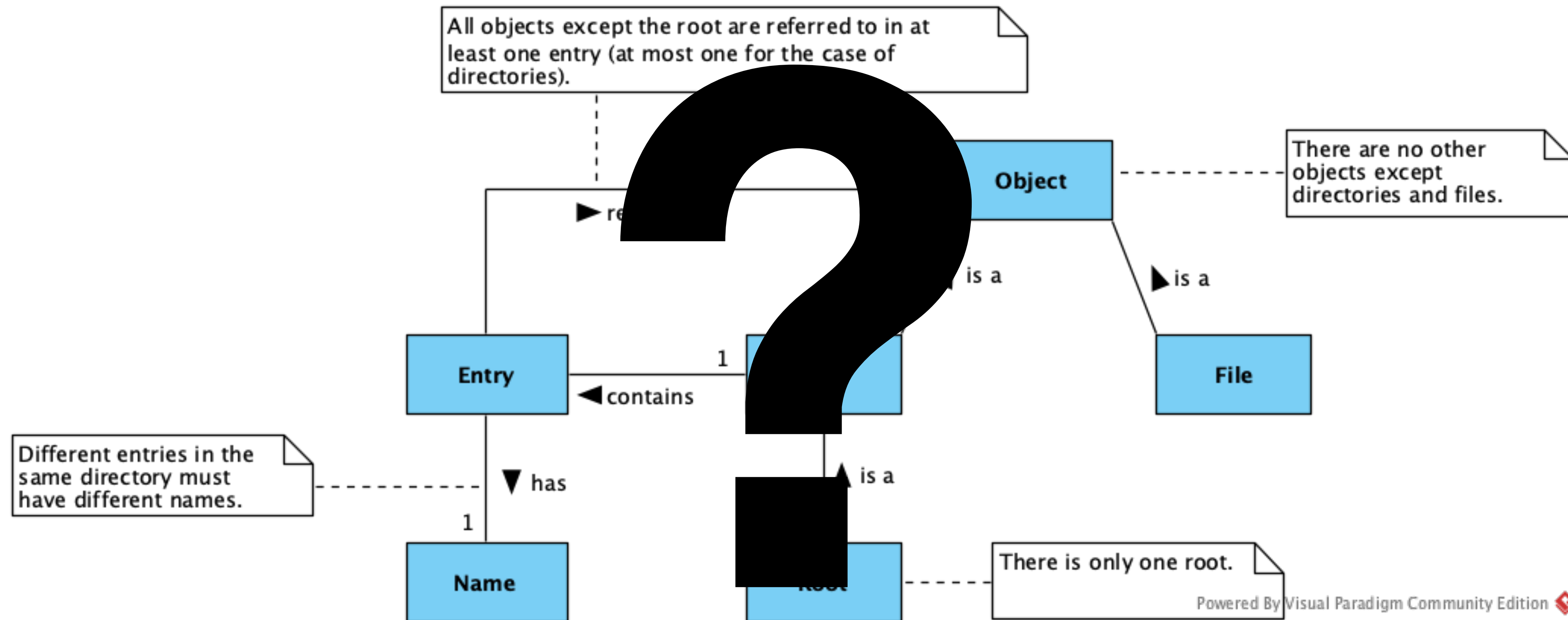
Daniel Jackson

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



–*Daniel Jackson*

Domain modeling with Alloy



First-order logic

Signatures

- Unary predicates are known as *signatures* and are declared with **sig**
- Signatures are inhabited by *atoms* from a finite *domain* of discourse
- Signatures can be top-level, extensions, or subsets
- Top-level and extension signatures are disjoint
- Signatures can be abstract, only containing atoms in the extensions
- Signatures can have a multiplicity (**lone**, **some**, **one**)

Top-level signatures

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

Object $\subseteq D$

Entry $\subseteq D$

Name $\subseteq D$

$\forall x. \neg(\text{Object}(x) \wedge \text{Entry}(x))$

$\forall x. \neg(\text{Object}(x) \wedge \text{Name}(x))$

$\forall x. \neg(\text{Name}(x) \wedge \text{Entry}(x))$

Extension signatures

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

$\text{Dir} \subseteq D$

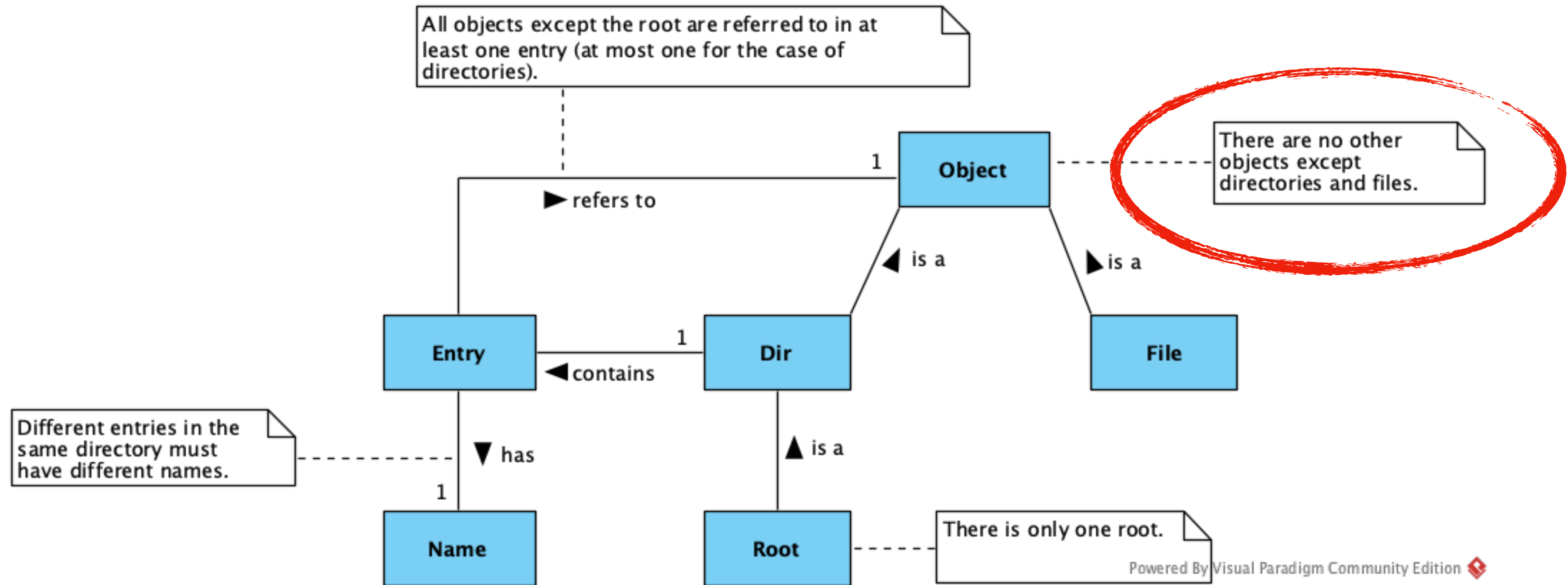
$\text{File} \subseteq D$

$\forall x. \text{File}(x) \rightarrow \text{Object}(x)$

$\forall x. \text{Dir}(x) \rightarrow \text{Object}(x)$

$\forall x. \neg(\text{File}(x) \wedge \text{Dir}(x))$

Object is abstract



Abstract signatures

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

$$\forall x : \text{Object} . \text{Dir}(x) \vee \text{File}(x)$$

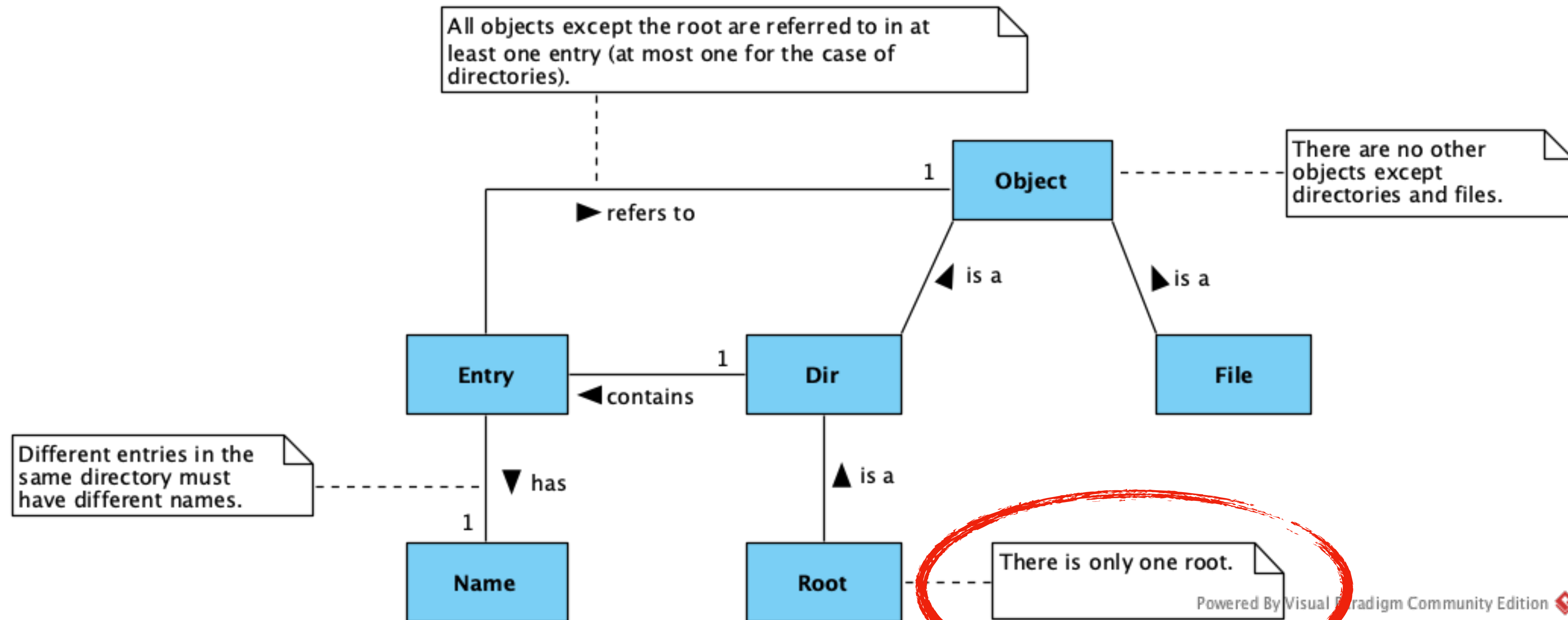
Subset signatures

sig Root **in** Dir {}

Root $\subseteq D$

$\forall x. \text{Root}(x) \rightarrow \text{Dir}(x)$

There is only one root



Signature multiplicities

one sig Root **in** Dir {}

$\exists x . \text{Root}(x)$

$\forall x, y : \text{Root} . x = y$

Fields

- Predicates of arity 2 or more are known as *fields*
- Fields are inhabited by *tuples* of atoms
- Must be declared inside the *domain* signature
- Multiplicities (**set**, **lone**, **some**, **one**) can be imposed on the targets
- If no multiplicity is imposed the default is **one**

Fields

```
sig Dir {  
  contains : set Entry  
}
```

contains $\subseteq D \times D$

$\forall x, y. \text{contains}(x, y) \rightarrow \text{Dir}(x) \wedge \text{Entry}(y)$

Fields

```
sig Entry {  
  refersTo : one Object,  
  has : one Name  
}
```

$\text{refersTo} \subseteq D \times D$

$\forall x, y . \text{refersTo}(x, y) \rightarrow \text{Entry}(x) \wedge \text{Object}(y)$

$\forall x : \text{Entry} . \exists y . \text{refersTo}(x, y)$

$\forall x, y, z . \text{refersTo}(x, y) \wedge \text{refersTo}(x, z) \rightarrow y = z$

$\text{has} \subseteq D \times D$

$\forall x, y . \text{has}(x, y) \rightarrow \text{Entry}(x) \wedge \text{Name}(y)$

$\forall x : \text{Entry} . \exists y . \text{has}(x, y)$

$\forall x, y, z . \text{has}(x, y) \wedge \text{has}(x, z) \rightarrow y = z$

Facts

- *Facts* specify assumptions

```
fact {  $\varphi$  }
```

- Facts can be named

```
fact Name {  $\varphi$  }
```

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

```
fact {  
     $\varphi$   
     $\psi$   
}
```

FOL vs Alloy

$\neg \phi$

$\phi \wedge \psi$

$\phi \vee \psi$

$\phi \rightarrow \psi$

$(\phi \wedge \psi) \vee (\neg \phi \wedge \theta)$

$\phi \leftrightarrow \psi$

! ϕ

$\phi \ \&\& \ \psi$

$\phi \ || \ \psi$

$\phi \ ==> \ \psi$

$\phi \ ==> \ \psi \ \mathbf{else} \ \theta$

$\phi \ <=> \ \psi$

FOL vs Alloy

$\neg\phi$

$\phi \wedge \psi$

$\phi \vee \psi$

$\phi \rightarrow \psi$

$(\phi \wedge \psi) \vee (\neg\phi \wedge \theta)$

$\phi \leftrightarrow \psi$

not ϕ

ϕ **and** ψ

ϕ **or** ψ

ϕ **implies** ψ

ϕ **implies** ψ **else** θ

ϕ **iff** ψ

FOL vs Alloy

$x = y$

$A(x)$

$R(x, y)$

$\forall x. A(x) \rightarrow \phi$

$\exists x. A(x) \wedge \phi$

$x = y$

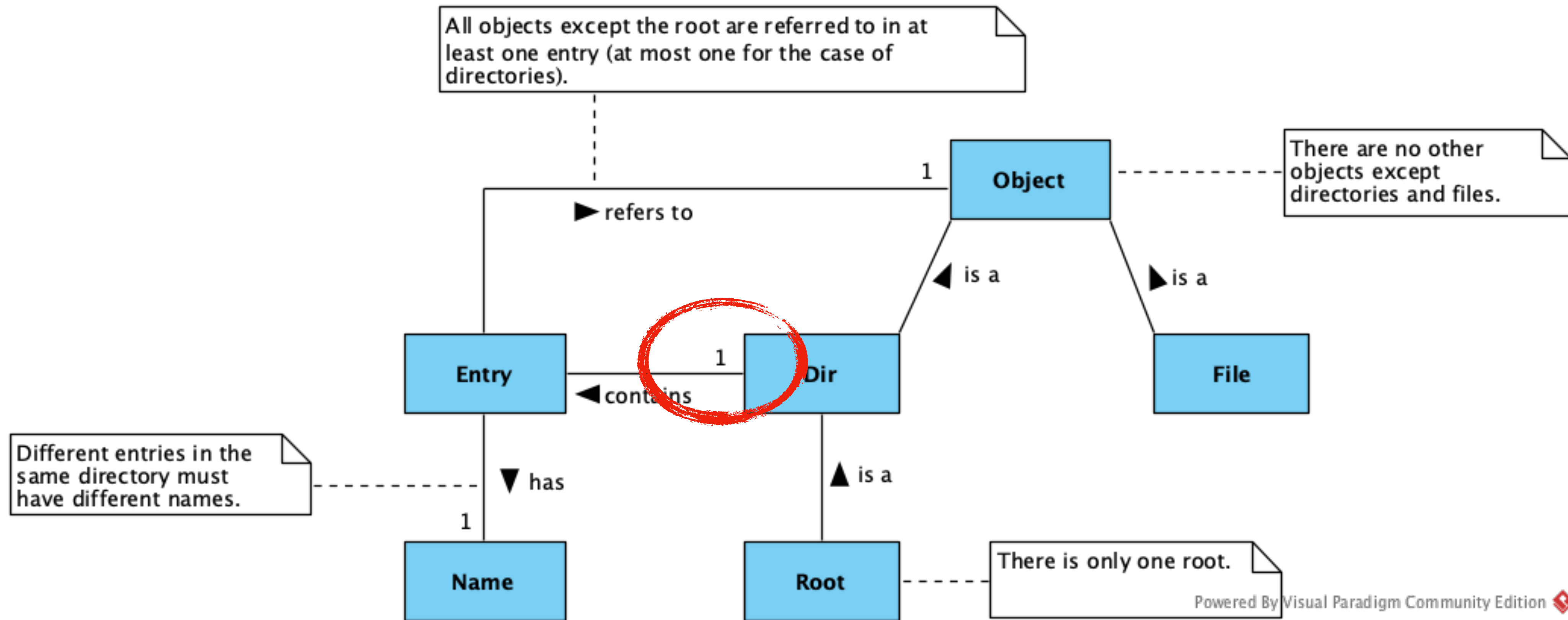
x **in** A

$x \rightarrow y$ **in** R

all $x : A \mid \phi$

some $x : A \mid \phi$

Each entry is contained in one directory



Each entry is contained in one directory

```
fact {  
  // Each entry is contained in one directory  
  all x : Entry | some y : Dir | y->x in contains  
  all x : Entry, y,z : Dir {  
    y->x in contains and z->x in contains implies y = z  
  }  
}
```

Commands

- Alloy has two types of analysis *commands*:
 - **run** { φ } asks for an *example* that satisfies all facts and φ
 - **check** { φ } asks for a *counter-example* that satisfies all facts but 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 first-order structures
- In Alloy first-order structures are known as *instances*
- An instance is a valuation to all the signatures and fields
- 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)
- By default instances are depicted as graphs

Scopes

- To ensure decidability commands have a *scope*
- The scope imposes a limit on the size of the (finite) domain 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!!!
- Anecdotal evidence suggests that most invalid assertions (or consistent predicates) can be refuted (or witnessed) with a small scope

Atoms

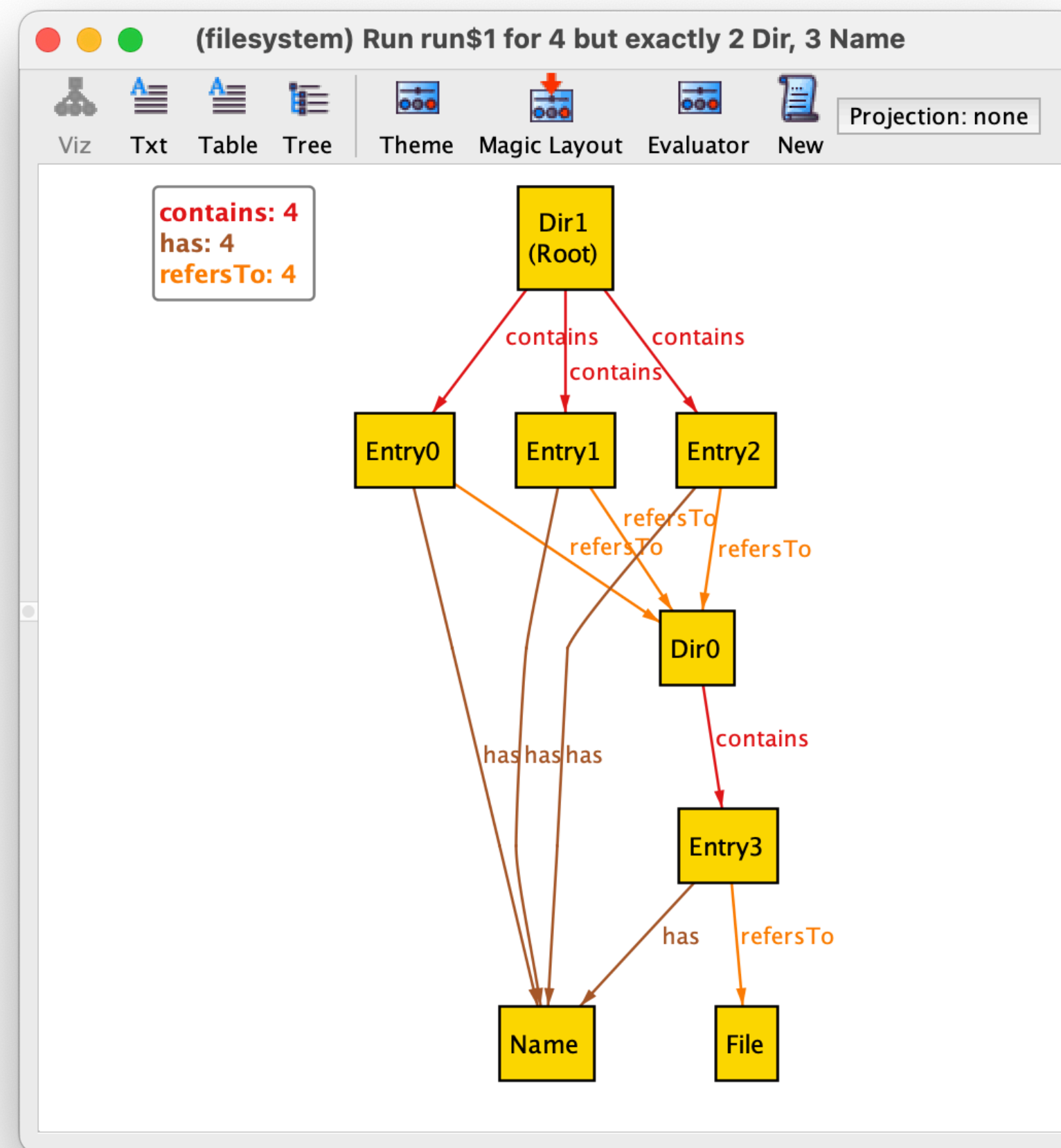
- The universe of discourse contains *atoms*
- Atoms are *uninterpreted* (no semantics)
- Named automatically according to the respective signatures
- Two instances are *isomorphic* (or *symmetric*) if they are equal modulo renaming
- The analysis implements a *symmetry breaking* mechanism to avoid returning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

A simple command

run {} for 4 but exactly 2 Dir, 3 Name

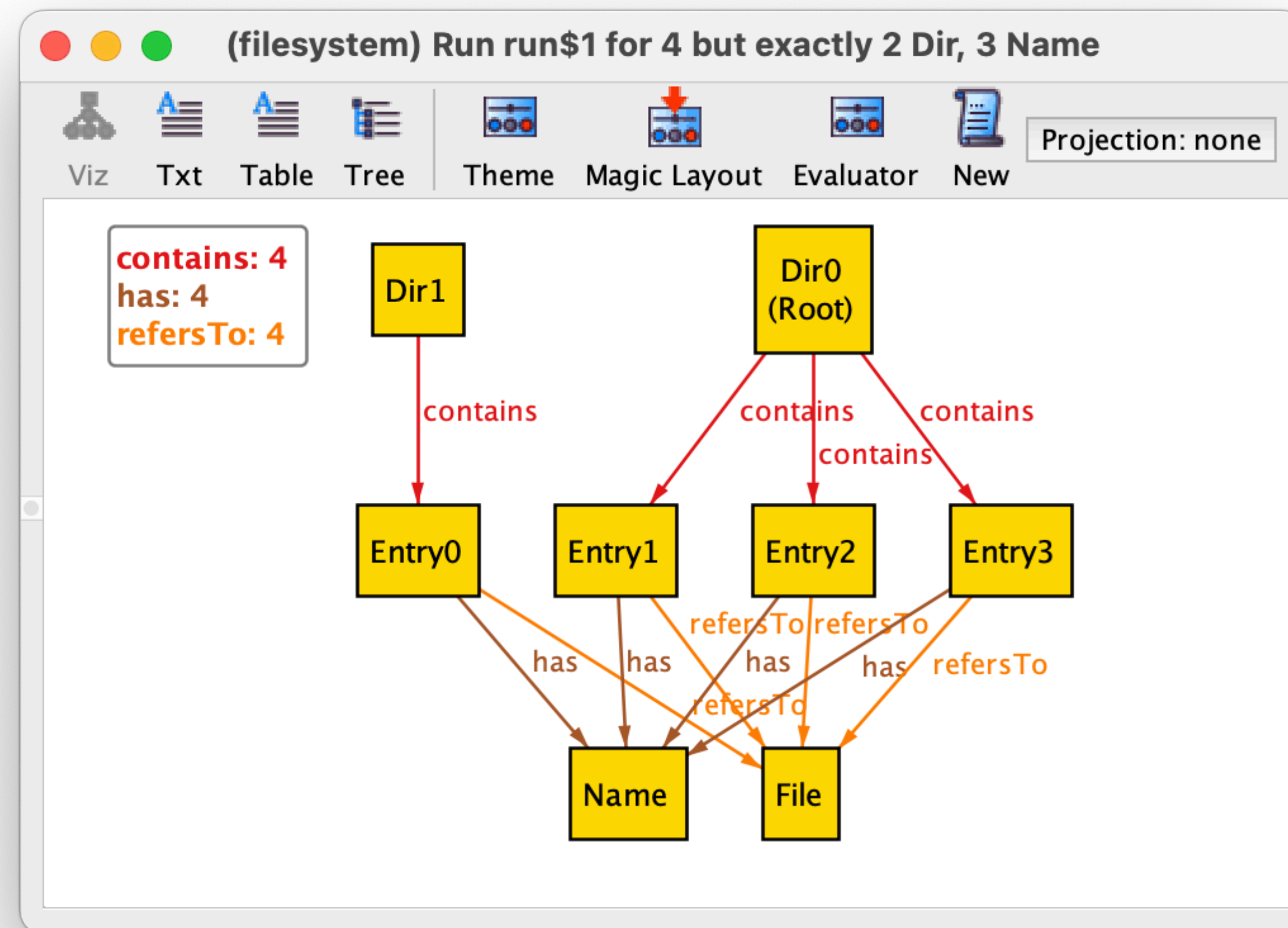
A simple command

run {} for 4 but exactly 2 Dir, 3 Name

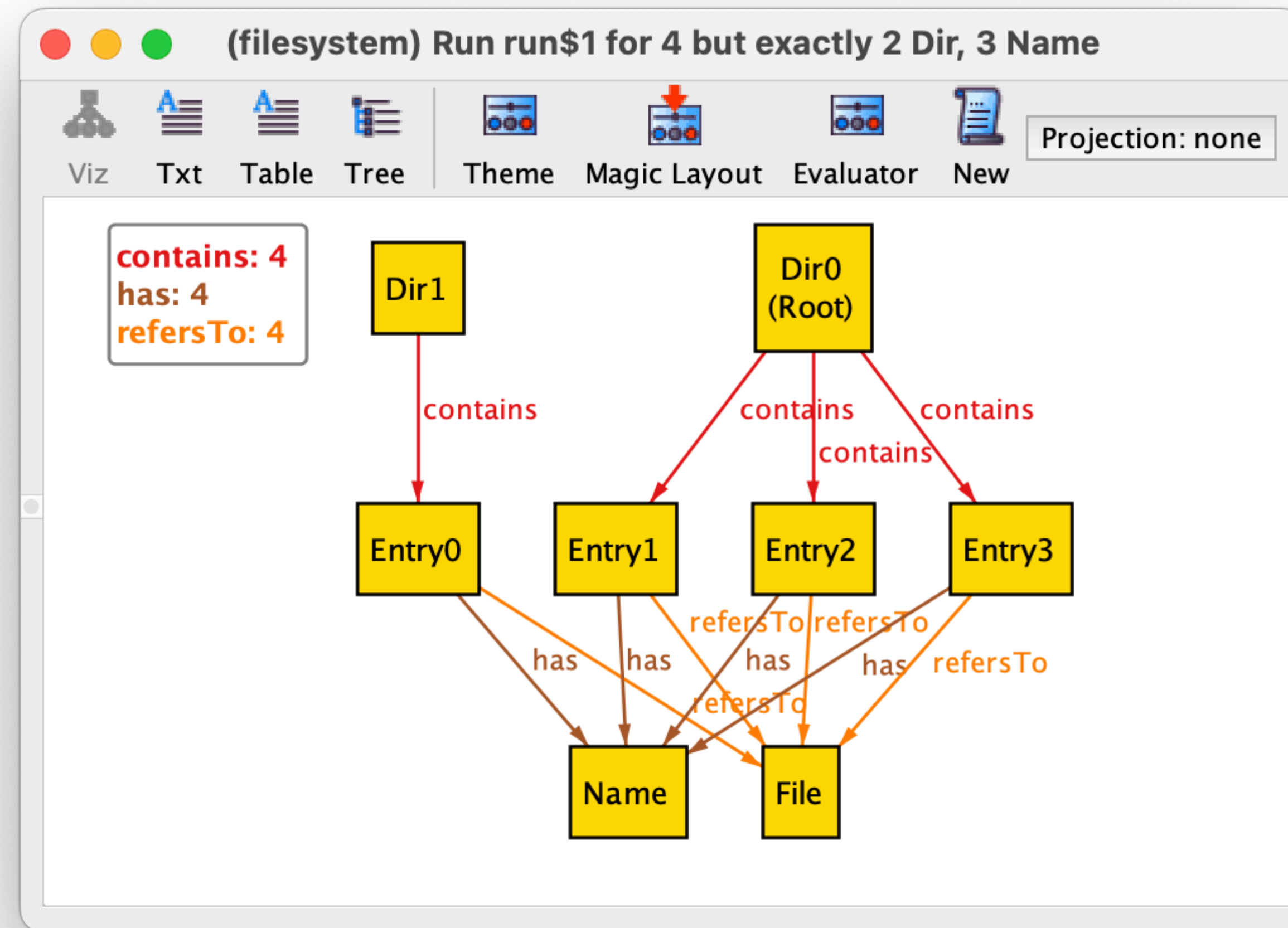


A simple command

run {} for 4 but exactly 2 Dir, 3 Name



Instances as graphs



Instances as relations

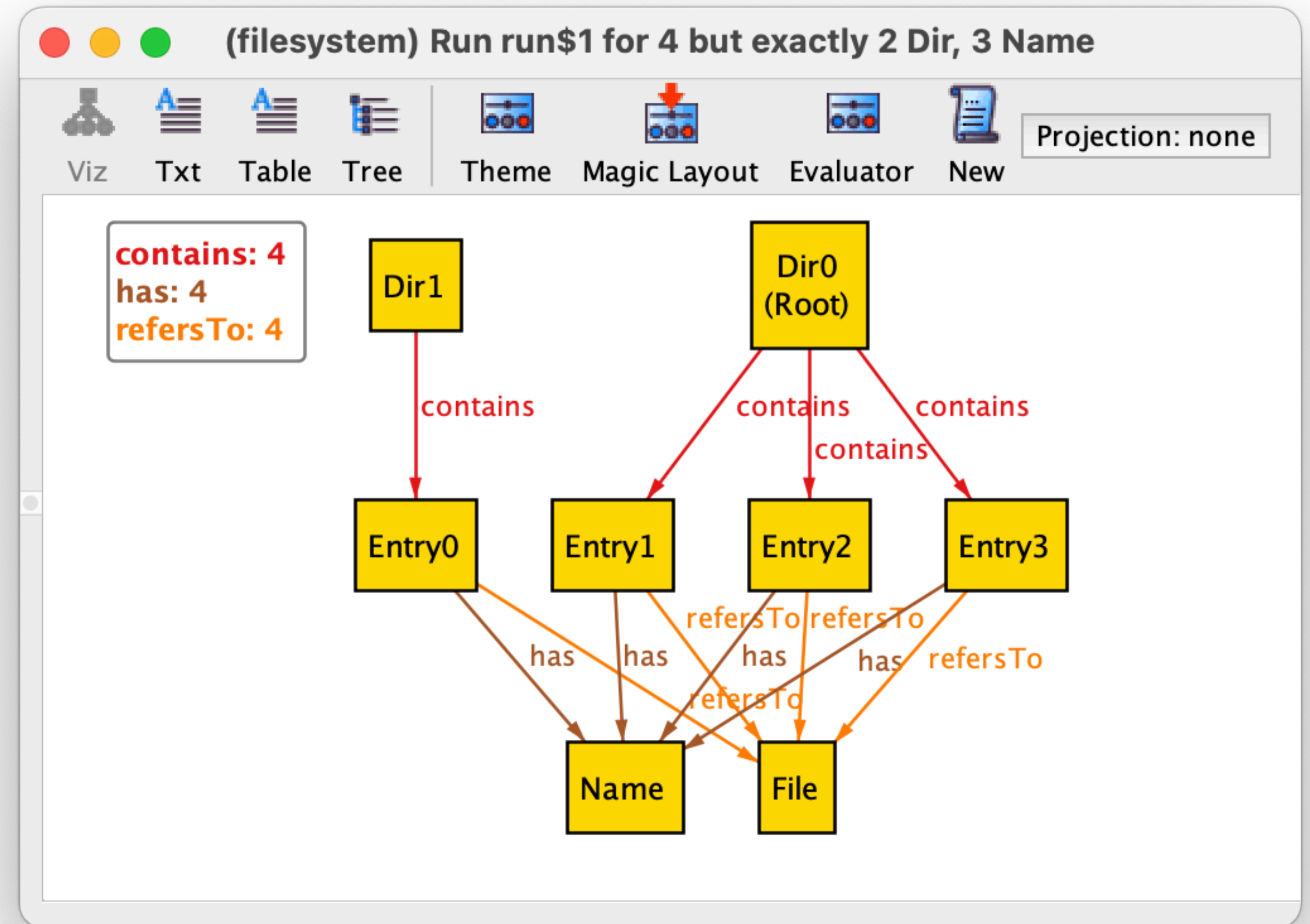
```
Object    = { (Dir0), (Dir1), (File) }
Dir       = { (Dir0), (Dir1) }
File      = { (File) }
Root      = { (Dir0) }
Entry     = { (Entry0), (Entry1), (Entry2), (Entry3) }
Name      = { (Name) }
contains  = { (Dir1, Entry0), (Dir0, Entry1), (Dir0, Entry2), (Dir0, Entry3) }
refersTo  = { (Entry0, File), (Entry1, File), (Entry2, File), (Entry3, File) }
has       = { (Entry0, Name), (Entry1, Name), (Entry2, Name), (Entry3, Name) }
```

Instances as tables

Object	Dir	Root	File	Name	Entry
Dir0	Dir0	Dir0	File	Name	Entry0
Dir1	Dir1				Entry1
File					Entry2
					Entry3

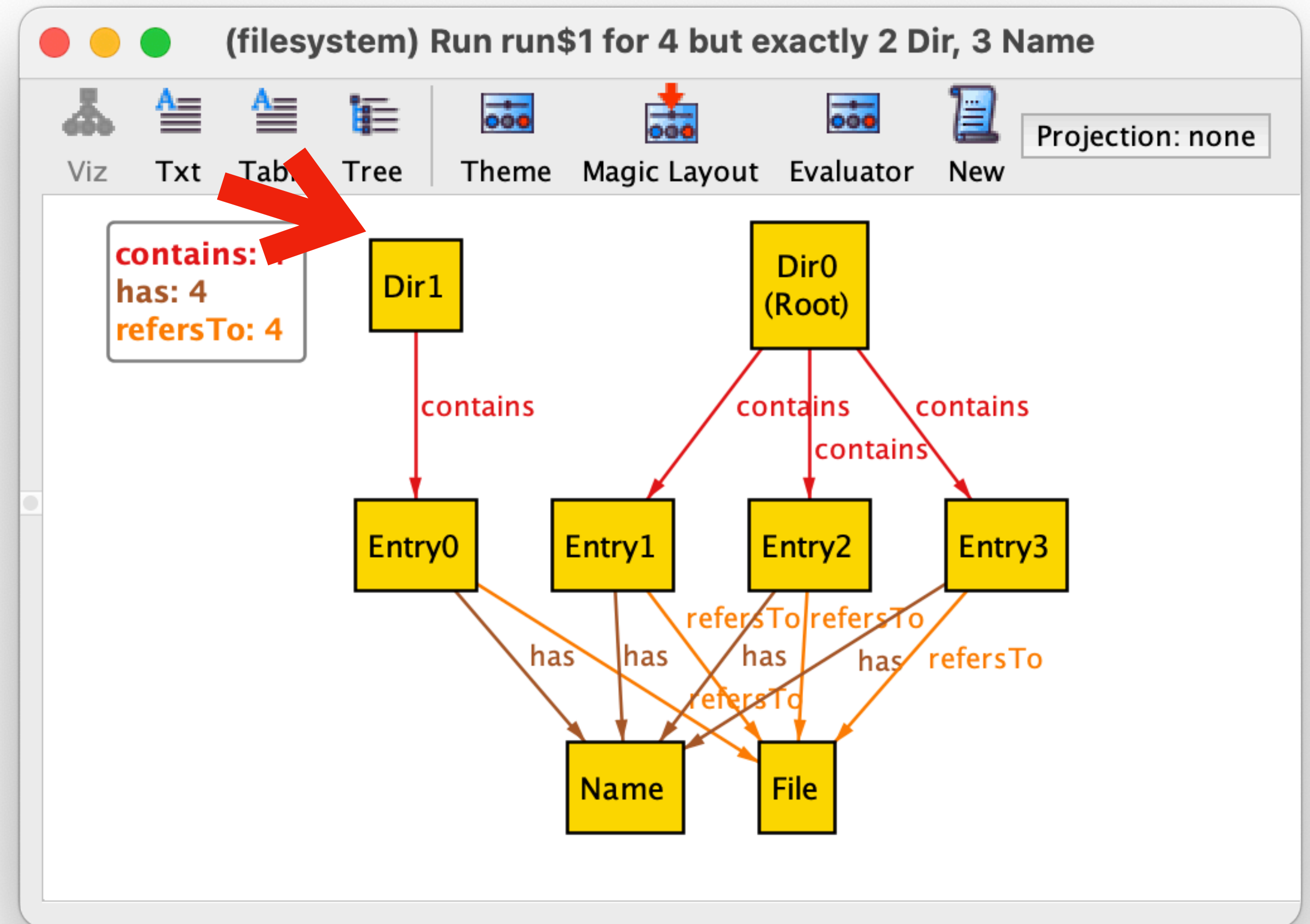
contains		refersTo		has	
Dir1	Entry0	Entry0	File	Entry0	Name
Dir0	Entry1	Entry1	File	Entry1	Name
Dir0	Entry2	Entry2	File	Entry2	Name
Dir0	Entry3	Entry3	File	Entry3	Name

Additional requirements



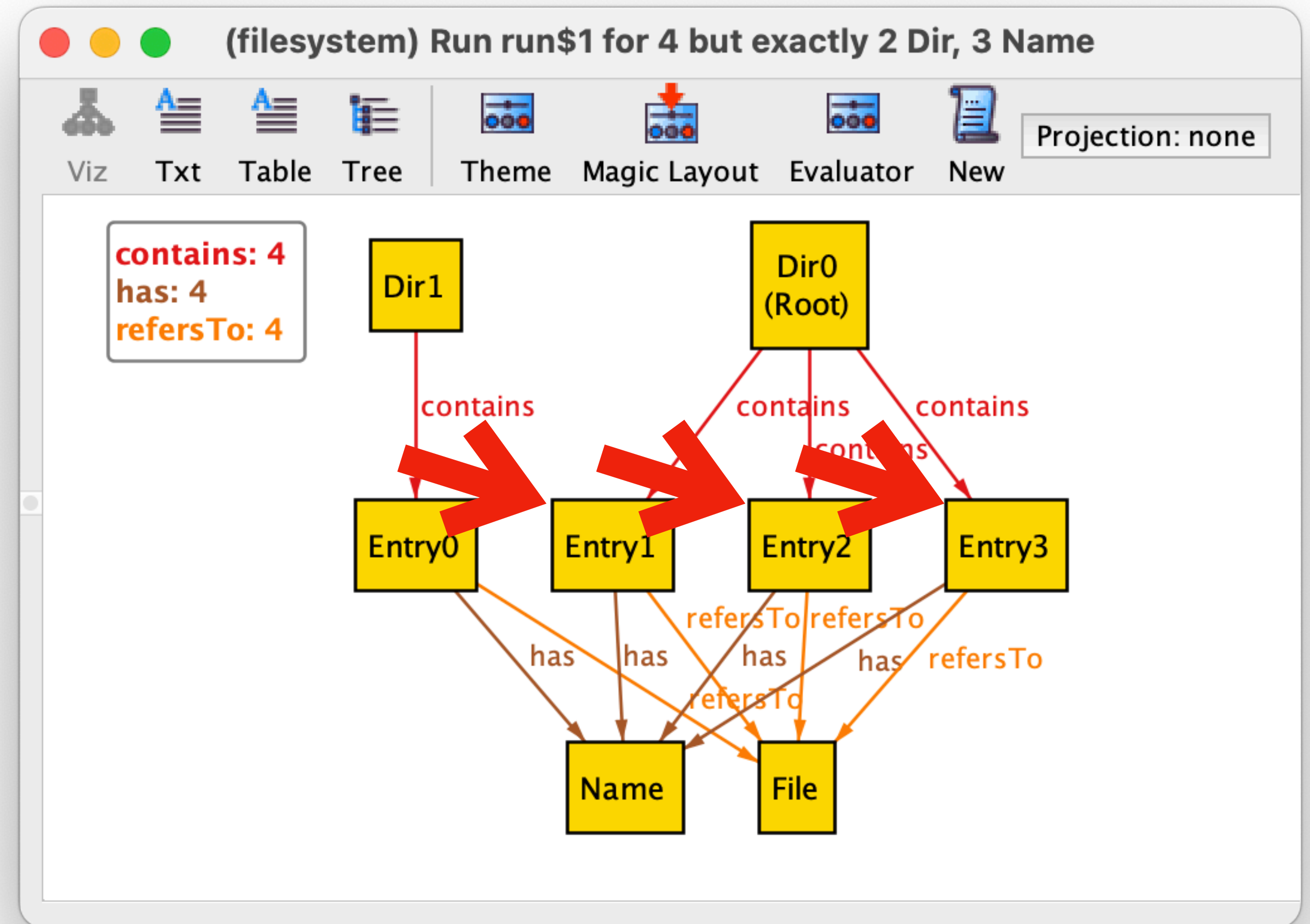
Additional requirements

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



Additional requirements

- 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



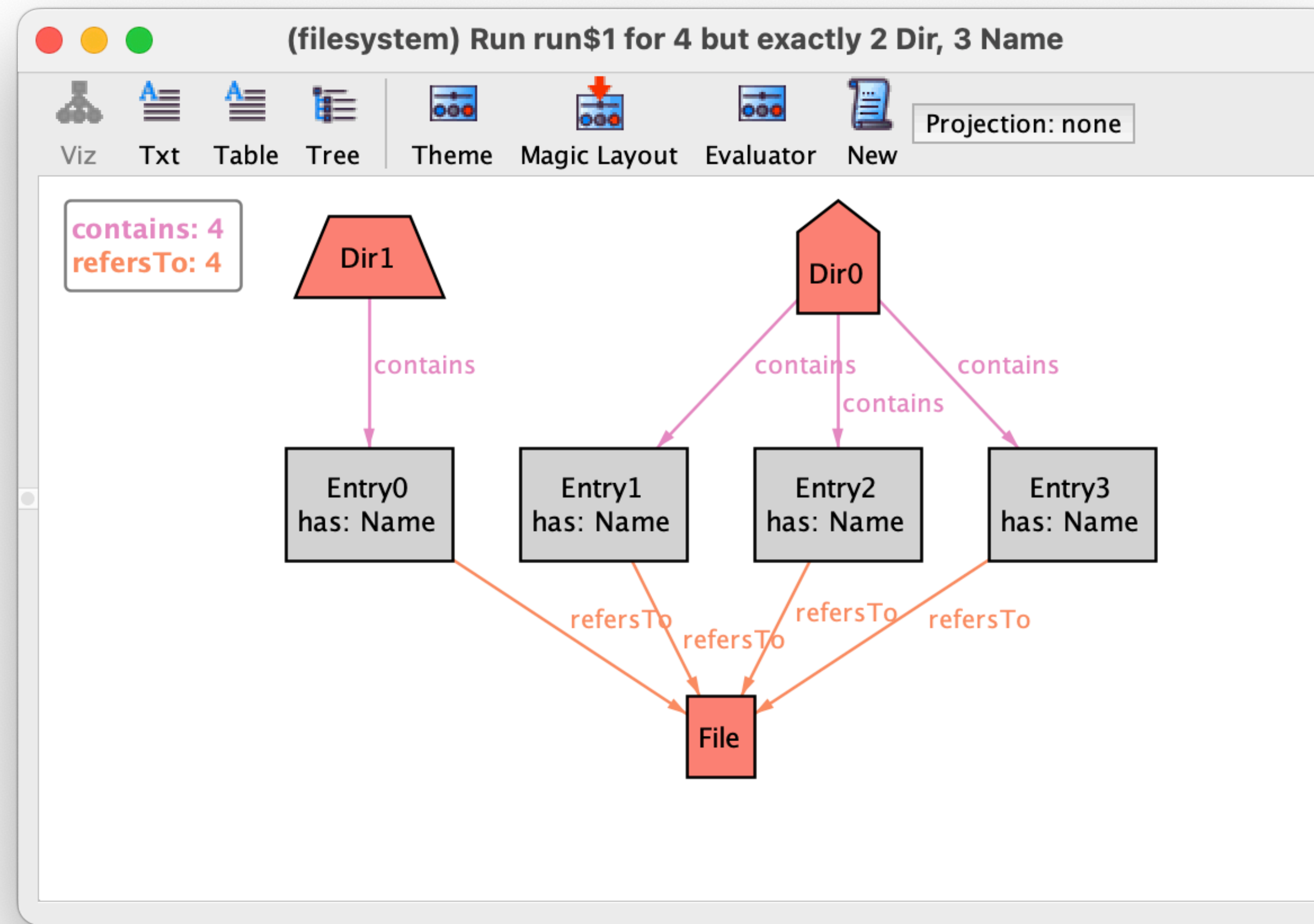
Themes

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

Theme customization



Theme customization



Requirements breakdown

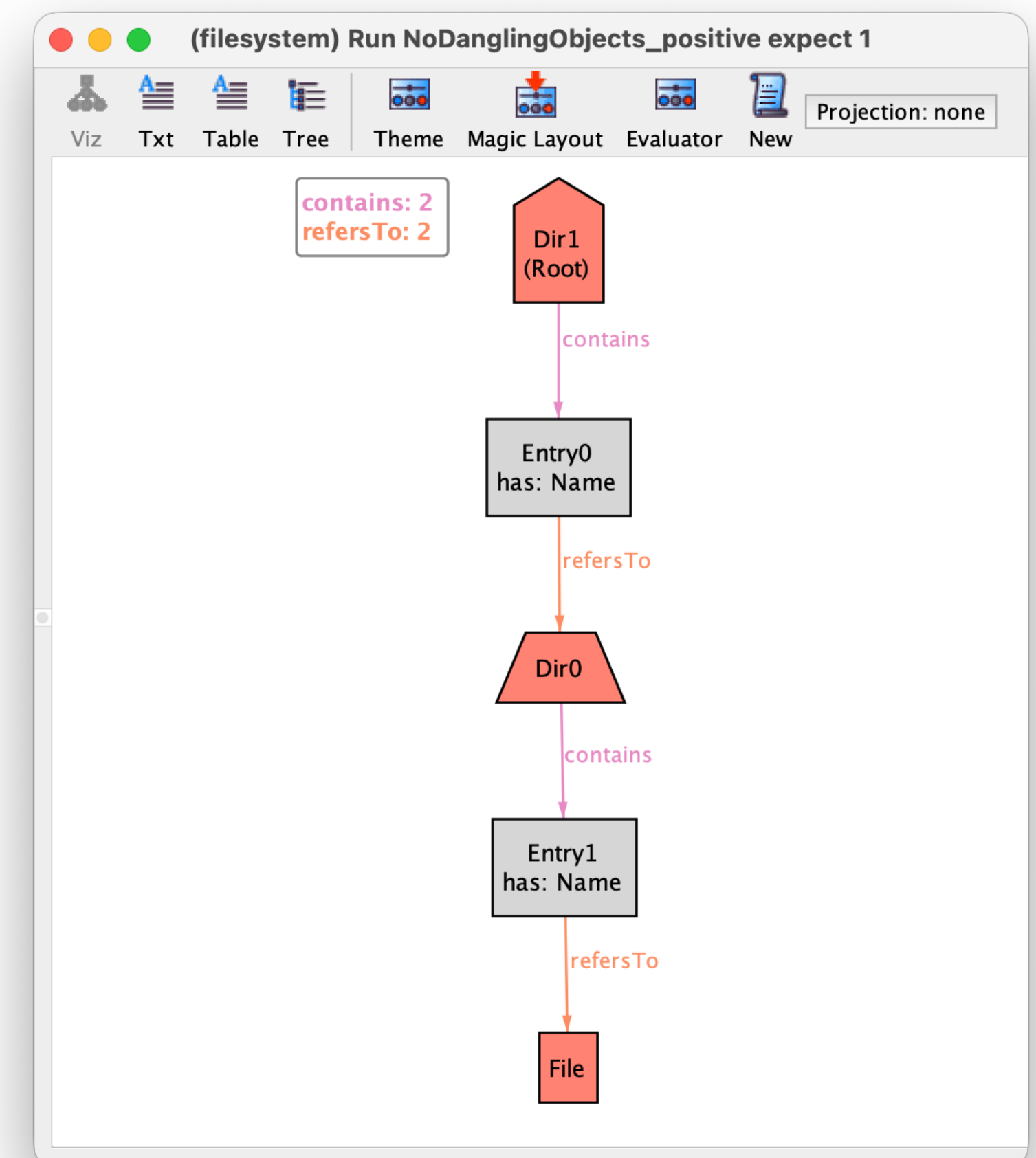
- All objects except the root are referred to in at least one entry (at most one for the case of directories)
 - All directories are referred to in at most one entry
 - The root is not referred in any entry
 - All non-root objects are referred to in at least one entry
- Different entries in a directory must have different names

Scenario driven specification

- Before specifying each requirement write some scenarios for validation
- Scenarios can be specified with **run** commands
- Include at least one positive and one negative scenario
 - Use **expect** keyword to document expectation
 - Verify that positive scenarios are SAT
 - Verify that negative scenarios are UNSAT
- Recheck all scenarios every time facts are added or changed

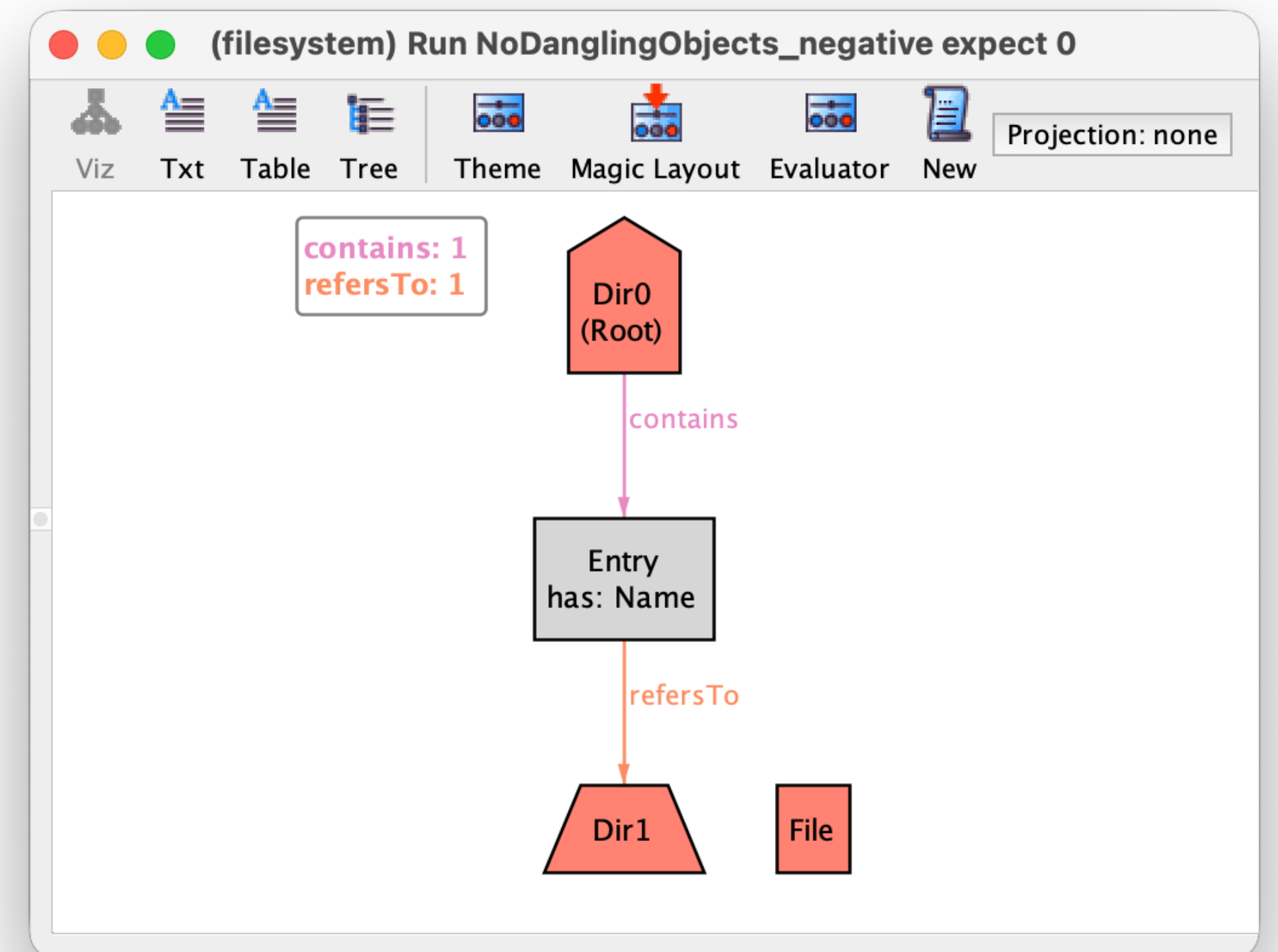
All non-root objects are referred to in at least one entry

```
run NoDanglingObjects_positive {  
  some r : Root, d : Dir, f : File, disj e1,e2 : Entry, n : Name {  
    Dir      = r + d  
    Root     = r  
    File     = f  
    Entry    = e1 + e2  
    Name     = n  
    contains = r->e1 + d->e2  
    refersTo = e1->d + e2->f  
    has      = e1->n + e2->n  
  }  
} expect 1
```



All non-root objects are referred to in at least one entry

```
run NoDanglingObjects_negative {  
  some r : Root, d : Dir, f : File, e : Entry, n : Name {  
    Dir      = r + d  
    Root     = r  
    File     = f  
    Entry    = e  
    Name     = n  
    contains = r->e  
    refersTo = e->d  
    has      = e->n  
  }  
} expect 0
```



**All non-root objects are referred to
in at least one entry**

```
fact NoDanglingObjects {  
    all x : Object | x not in Root implies some y : Entry | y->x in refersTo  
}
```

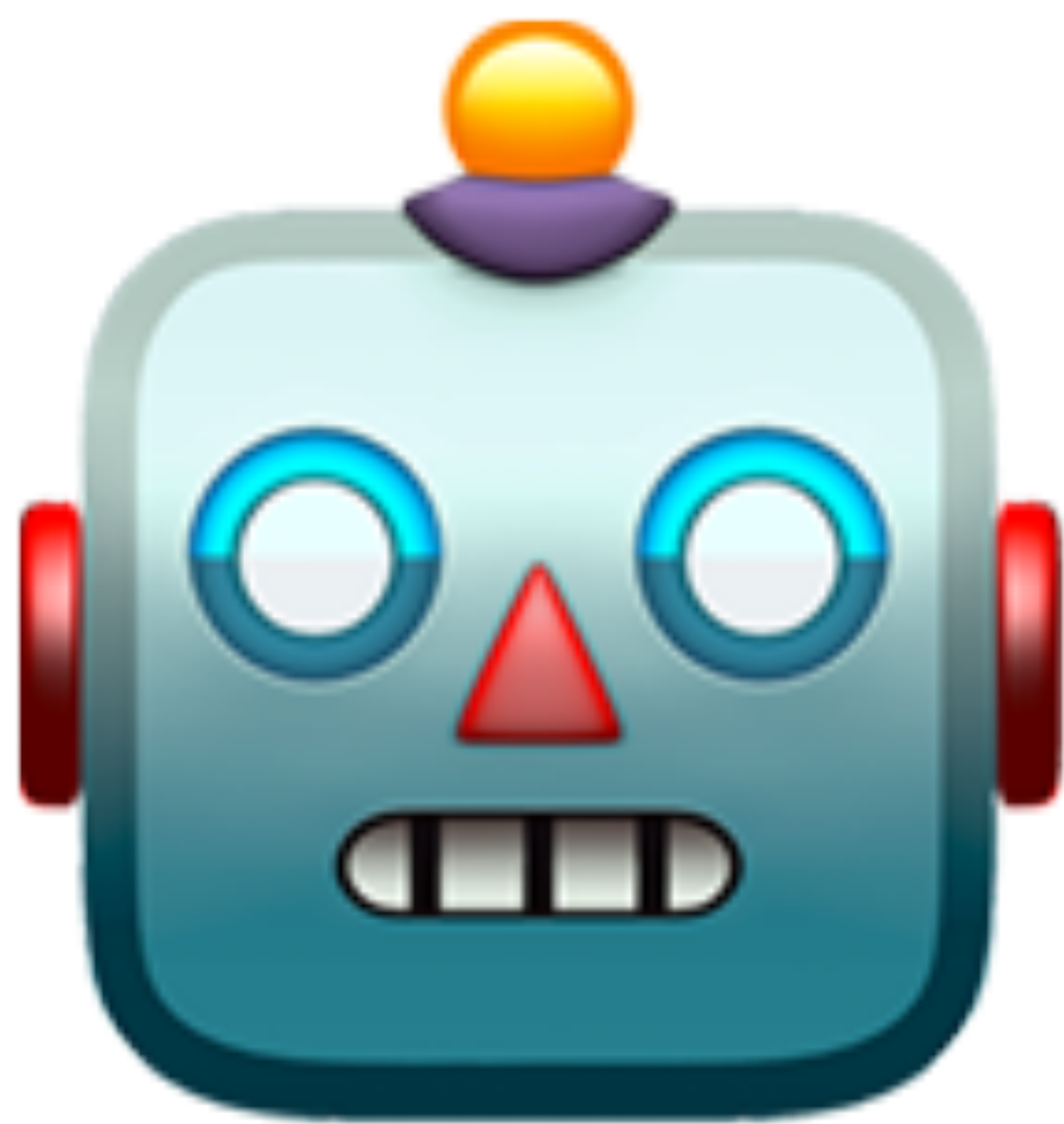
The remaining requirements

```
// All directories are referred to in at most one entry
fact NoSharedDirs {
    all x : Dir, y,z : Entry | y->x in refersTo and z->x in refersTo implies y = z
}

// The root is not referred in any entry
fact RootIsDangling {
    all x : Entry, y : Root | x->y not in refersTo
}

// Different entries in a directory must have different names
fact UniqueNames {
    all x : Dir, y,z : Entry, w : Name {
        x->y in contains and x->z in contains and y->w in has and z->w in has implies y = z
    }
}
```





Specification Engineering with LLMs

- Scenario generation
 - Safe, because user can easily review the scenarios
- Autoformalization
 - Difficult to review the result and dangerous if trusted blindly
 - Risk can be reduced if paired with scenarios, equivalence checking between alternatives (ideally proposed by different models), ...
 - But there will always be some risk, we need to understand the specs!

Relational logic