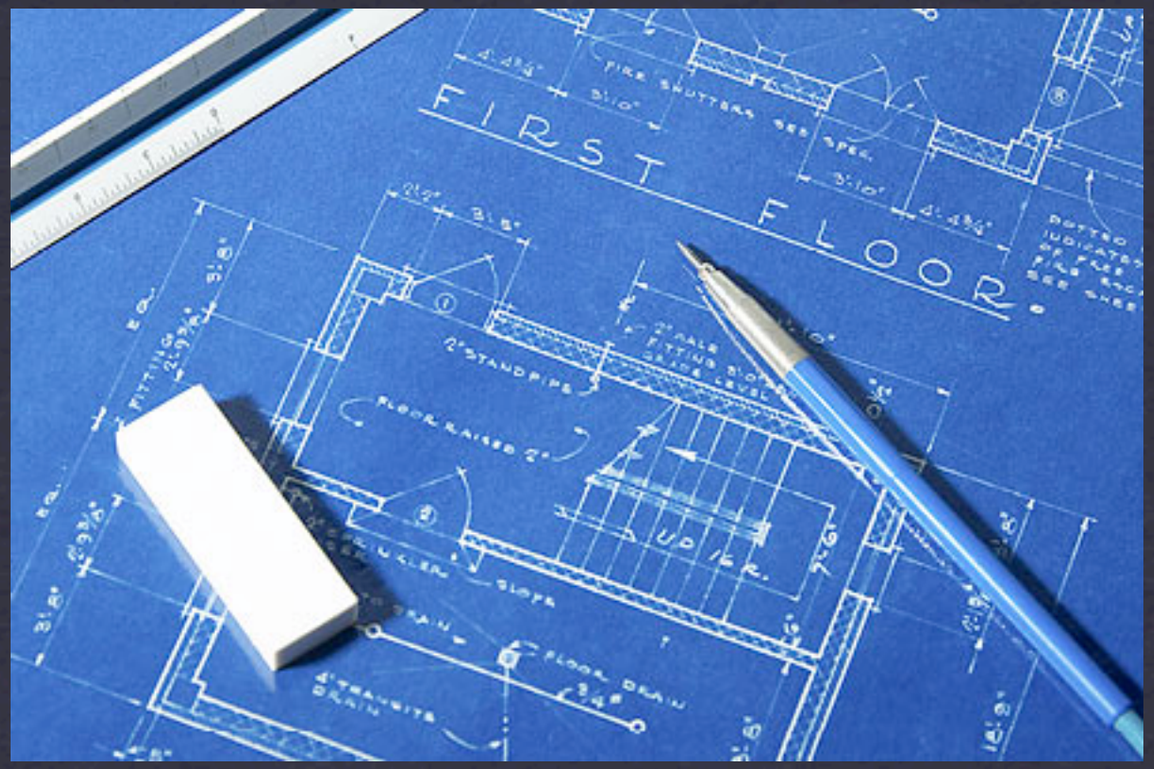
Mapping between Alloy specifications and database implementations



Hugo Pacheco

CIC'09

Universidade do Minho, 7 May 2009

Motivation

- which correspondence exists?
  - *“in Alloy, everything is a relation”*
  - and the same happens in relational databases
- what for?
  - generate implementations
  - reverse legacy systems

The Alloy language

- signatures (sets) **sig A {}** **one sig A {}** **lone sig A {}** **some sig A {}**
  - **sig B in A {}** **abstract sig A {}** **sig B extends A {}**
- fields (relations) **sig A { r : set B }**
  - **sig A { r : lone B }** **sig A { r : set B, s : r -> C}**
- predicates (first-order logic formulas)
- facts (predicates that always hold)

Filesystem example

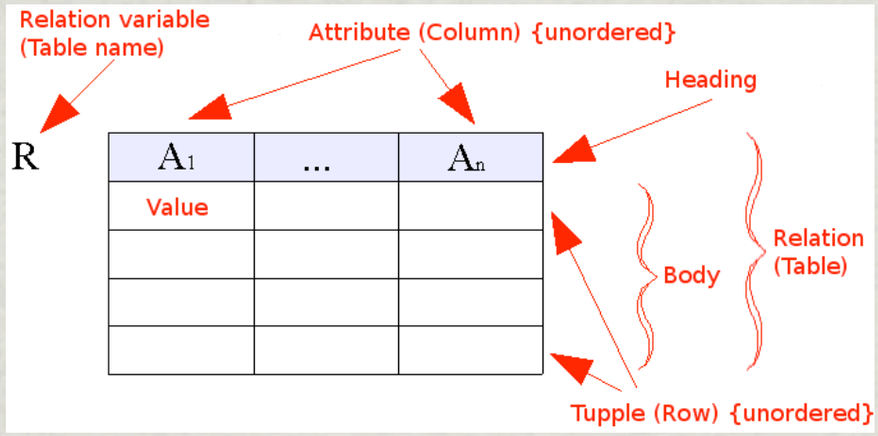
```
abstract sig Object {}
sig Dir extends Object {}
sig File extends Object {}
sig FS {
        objects : set Object,
        parent : (objects - root) -> one (Dir & objects),
        root : one (Dir & objects)
}

fact {
        all fs : FS | fs.objects in *(fs.parent).(fs.root) //root reachability
}

pred cd [fs, fs' : FS, d : Dir] { … }
pred mv [fs, fs' : FS, o : Object, d : Dir] { … }
```

Relational model

- n-ary relations are subsets of the (commutative) cartesian product of n set domains
  - $R(A_1, \ldots, A_n)$
    - attribute names are distinct
    - attribute domains are sets



- projections are relations (remove duplicates)
  - $R[A_1]$ $R[A_1, A_3]$

Relational model

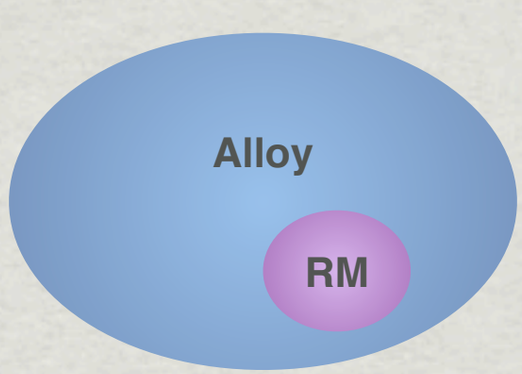
- first normal form (1NF)
  - *“each table directly and faithfully represents a relation”*
    - no row or column ordering, no nulls, no duplicate rows
- Alloy
  - all signatures and fields are first-order relations
  - Alloy models are in the 1NF (overlooking column ordering)

Relational constraints

- candidate keys (minimal superkeys)
- foreign keys are inclusion constraints
  - $\mathbf{R[A_1] \subseteq S[B_1]}$
  - $\mathbf{R[A_1, A_3] \subseteq S[B_2, B_3]}$
  - can only reference superkeys (many-to-one)

An Alloy subset

- we know that most of Alloy first-order logic operators cannot be mapped to our relational model



- we identify a proper subset of Alloy that is equivalent to our relational model

An Alloy subset

- signatures
  - inclusion
  - no extension (how to guarantee that 2 tables are disjoint?)
  - no abstract (how to guarantee that a table is the union of 2 other tables?)
- fields
  - constraints on fields must be expressed as facts

An Alloy subset

- facts (lhs **in** rhs)
  - lhs ::= relation (signature or field or permutation)
    - rhs ::= cartesian product of projections with multiplicities
    - projection ::= relational projection à la Alloy
    - multiplicity ::= **lone** or **set** (no **one** nor **some**)
- one non-syntactic restriction: references to non-keys

Refactoring the example

```
abstract sig Object {}
sig Dir extends Object {}
sig File extends Object {}
sig FS {
        objects : set Object,
        parent : (objects - root) -> one (Dir & objects),
        root : one (Dir & objects)
}



fact {
     all fs : FS | fs.objects in *(fs.parent).(fs.root) //root reachability
}
```

Refactoring the example

```
sig Object {}
sig Dir in Object {}
sig File in Object {}
sig FS {
        objects : set Object,
        parent : (objects - root) -> one (Dir & objects),
        root : one (Dir & objects)
}
pred objInv {
        Object in Dir + File    //abstract
        no Dir & File           //disjoint extensions
}
fact {
    all fs : FS | fs.objects in *(fs.parent).(fs.root) //root reachability
}
```

Refactoring the example

```
sig Object {}
sig Dir in Object {}
sig File in Object {}
sig FS {
        objects : set Object,
        parent : (objects - root) -> one (Dir & objects),
        root : one (Dir & objects)
}
pred objInv {
        Object in Dir + File    //abstract
        no Dir & File           //disjoint extensions
}
fact {
     all fs : FS | fs.objects in *(fs.parent).(fs.root) //root reachability
}
```

Refactoring the example

```
sig Object {}
sig Dir in Object {}
sig File in Object {}
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : Dir
}
pred objInv {
        Object in Dir + File    //abstract
        no Dir & File           //disjoint extensions
}
fact {
      all fs : FS | fs.objects in *(fs.parent).(fs.root) //root reachability
}
```

Refactoring the example

```
sig Object {}
sig Dir in Object {}
sig File in Object {}
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : Dir
}
pred objInv {
        Object in Dir + File    //abstract
        no Dir & File           //disjoint extensions
}
fact {
      all fs : FS | fs.objects in *(fs.parent).(fs.root) //root reachability
}
```

Refactoring the example

```
sig Object {}
sig Dir in Object {}
sig File in Object {}
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : Dir
}
pred objInv {
        Object in Dir + File    //abstract
        no Dir & File           //disjoint extensions
}
pred inv[fs : FS] {
     fs.objects in *(fs.parent).(fs.root) //root reachability
}
```

Refactoring the example

```
sig FS {
        objects : set Object,
        parent : (objects - root) -> one (Dir & objects),
        root : one (Dir & objects)
}
```

Refactoring the example

```
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : one (Dir & objects)
}
fact filesystem {
        parent in objects -> lone Dir
        {f : FS, d : Dir, o : Object | f -> o -> d in parent} in objects -> Object



}
pred filesystemInv[fs : FS] {
      fs.parent in (fs.objects - fs.root) -> one Dir
}
```

Refactoring the example

```
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : one (Dir & objects)
}
fact filesystem {
       parent in objects -> lone Dir
       {f : FS, d : Dir, o : Object| f -> o -> d in parent} in objects -> Object



}
pred filesystemInv[fs : FS] {
      fs.parent in (fs.objects - fs.root) -> one Dir
}
```

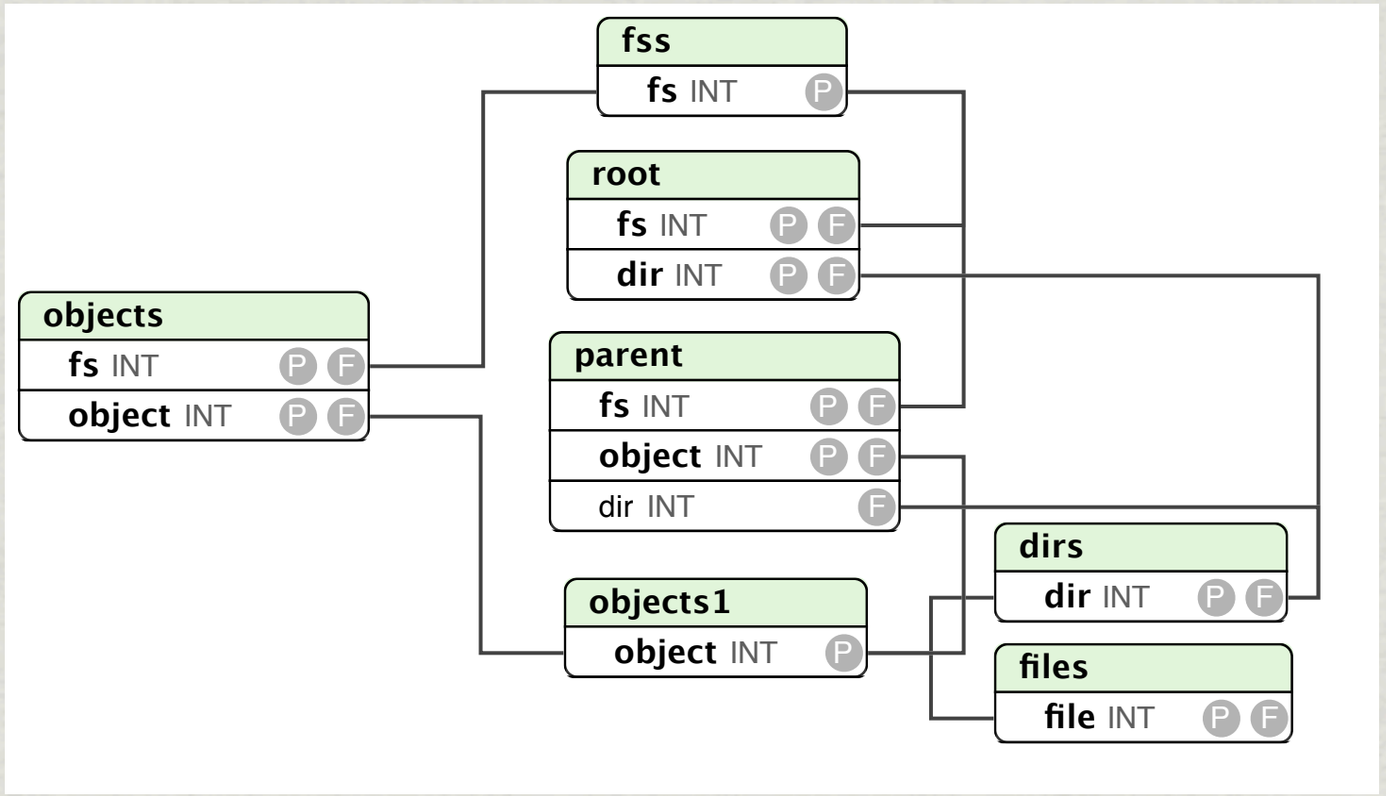
Refactoring the example

```
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : Dir
}
fact filesystem {
        parent in objects -> lone Dir
        {f : FS, d : Dir, o : Object | f -> o -> d in parent} in objects -> Object
        root in FS -> lone Dir
        FS in root.Dir
        root in objects
}
pred filesystemInv[fs : FS] {
      fs.parent in (fs.objects - fs.root) -> one Dir
}
```

Mapping the example

- create signature and field tables

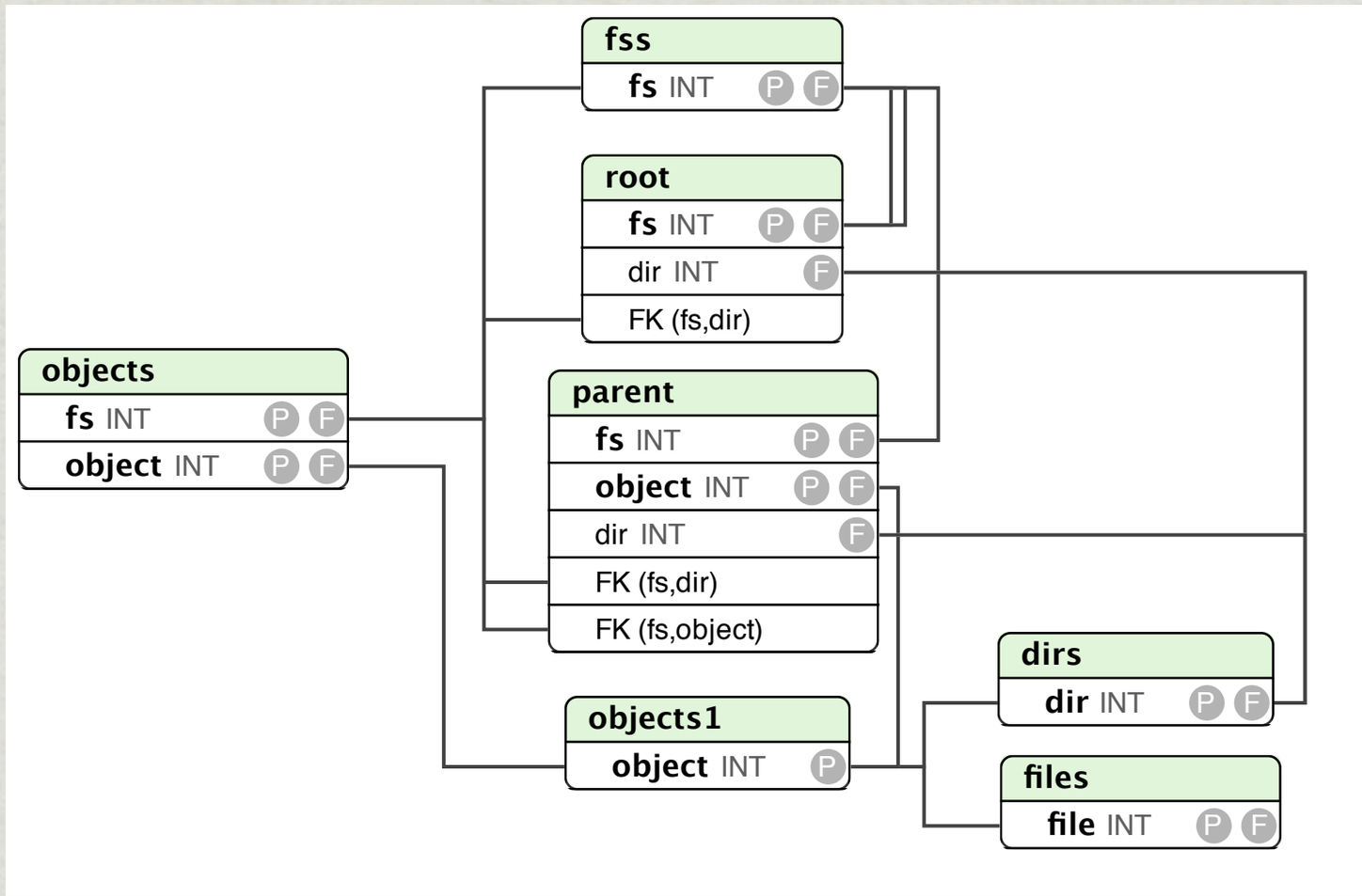
```
sig Object {}
sig Dir in Object {}
sig File in Object {}
sig FS {
        objects : set Object,
        parent : Object -> Dir,
        root : Dir
}
```



Mapping the example

- create facts keys and references

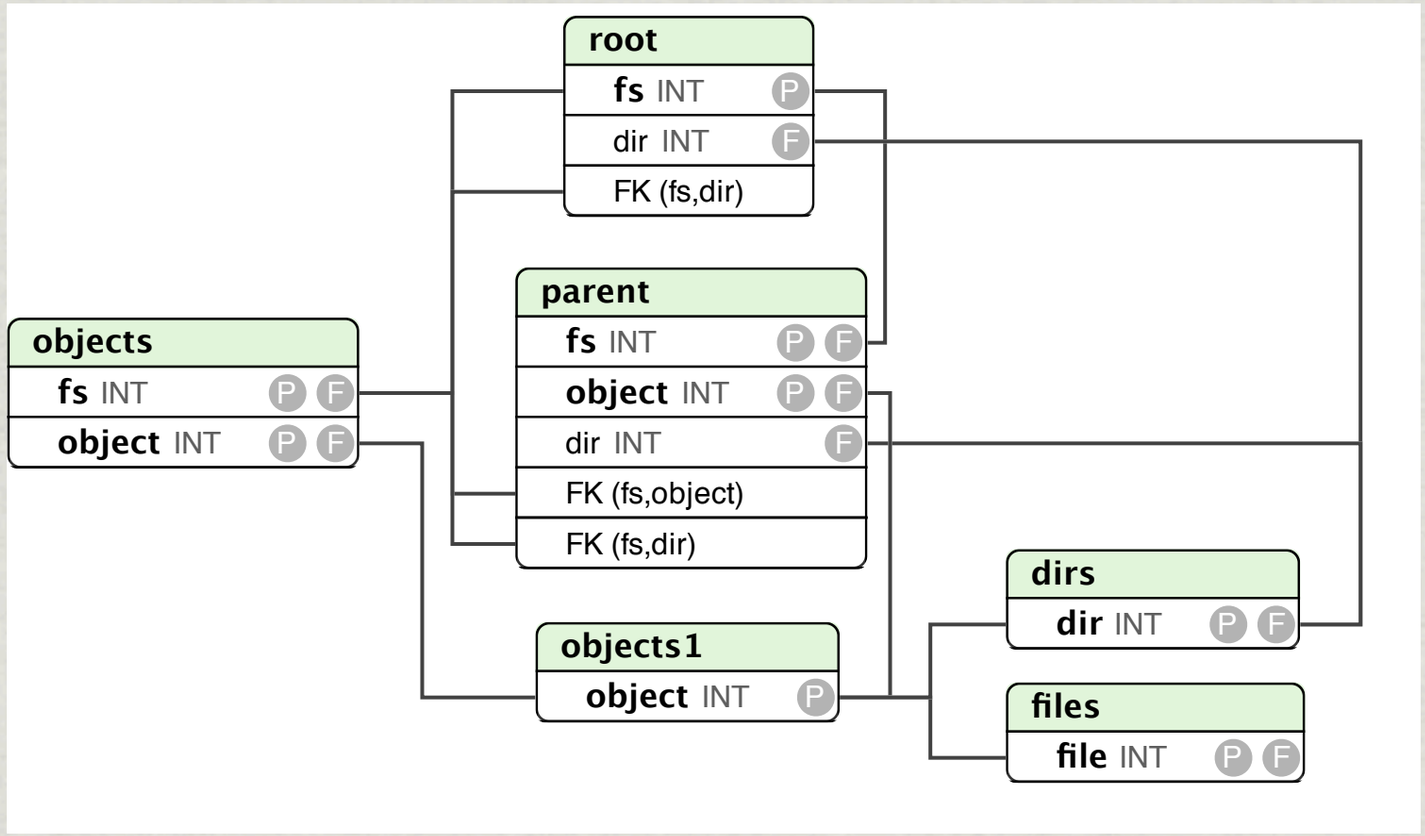
{f : FS, d : Dir, o : Object| f -> o -> d in parent} in objects -> Object
parent in objects -> lone Dir
root in FS -> lone Dir
FS in root.Dir
root in objects



Mapping the example

- merge the filesystems table

$$\mathbf{R}(\underline{\mathbf{A}},\mathbf{B}),\ \mathbf{S}(\underline{\mathbf{A}},\mathbf{C}) \Leftrightarrow \mathbf{RS}(\underline{\mathbf{A}},\mathbf{B},\mathbf{C})$$



Reversing the example

- create primary signatures (table columns)

```
sig FS {}
sig Object {}
```

- create subset signatures (unary tables)

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

Reversing the example

- create fields (tables) and multiplicities (keys)

```
sig FS {
        objects :  set Object,
        root : set Dir,
        parent : Object -> Dir
}
```

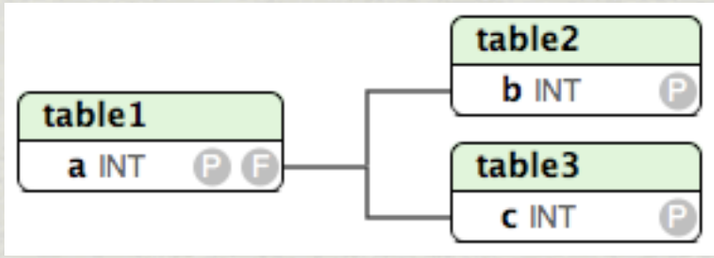
```
fact {
        root in FS -> lone Dir
        FS in root.Dir
        parent in FS -> Object -> lone Dir
}
```

- create facts (references)

```
fact {
        root in objects
        parent in objects -> object
        {f : FS, d : Dir, o : Object| f -> o -> d in parent} in objects -> Object
}
```

Multiple inheritance

- is it possible to reverse any database into a specification in our Alloy subset?
- what about this example?



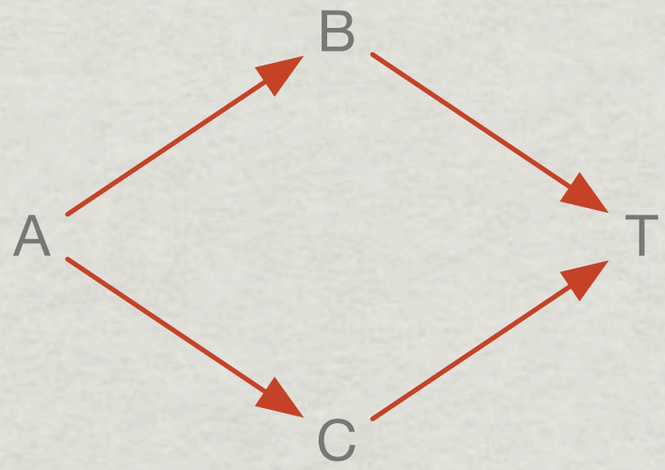
- problem: references to disjoint columns

```
sig B {}
sig C {}
sig A in B {}
fact { A in C }
```

Multiple inheritance

- which is the minimal extension to the subset?
  - create a “virtual” top signature (diamond)

```
abstract sig T {}
sig B in T {}
sig C in T {}
sig A in T {}
fact {
      A in B
      A in C
}
```



- … (forbidden for now)

Conclusions

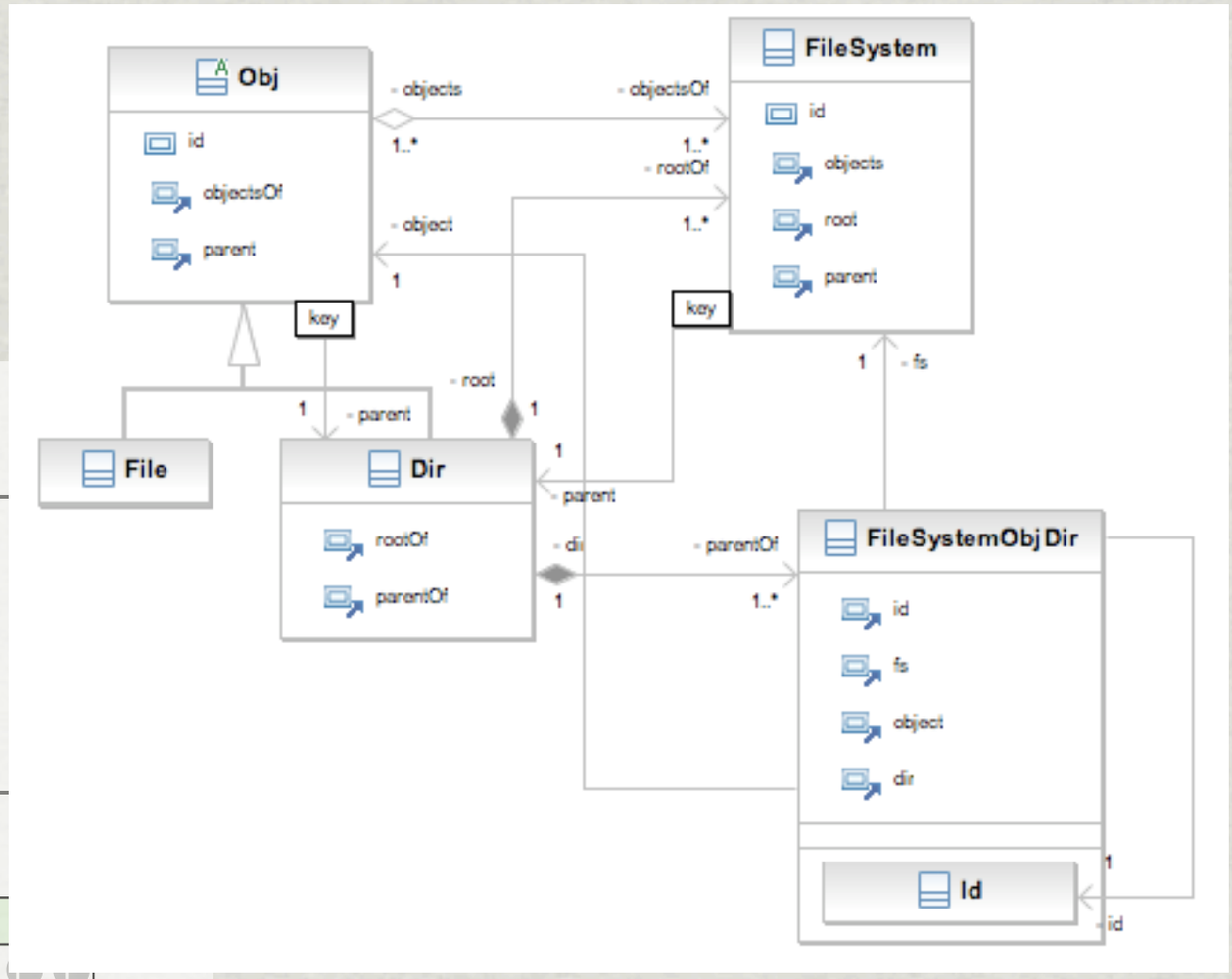
- we identified a subset of Alloy that is (informally) equivalent to a database model with keys and refs
- we presented how to map between those representations (prototype tool)
- future work:
  - can we cover more constraints?
  - Alloy refactoring system

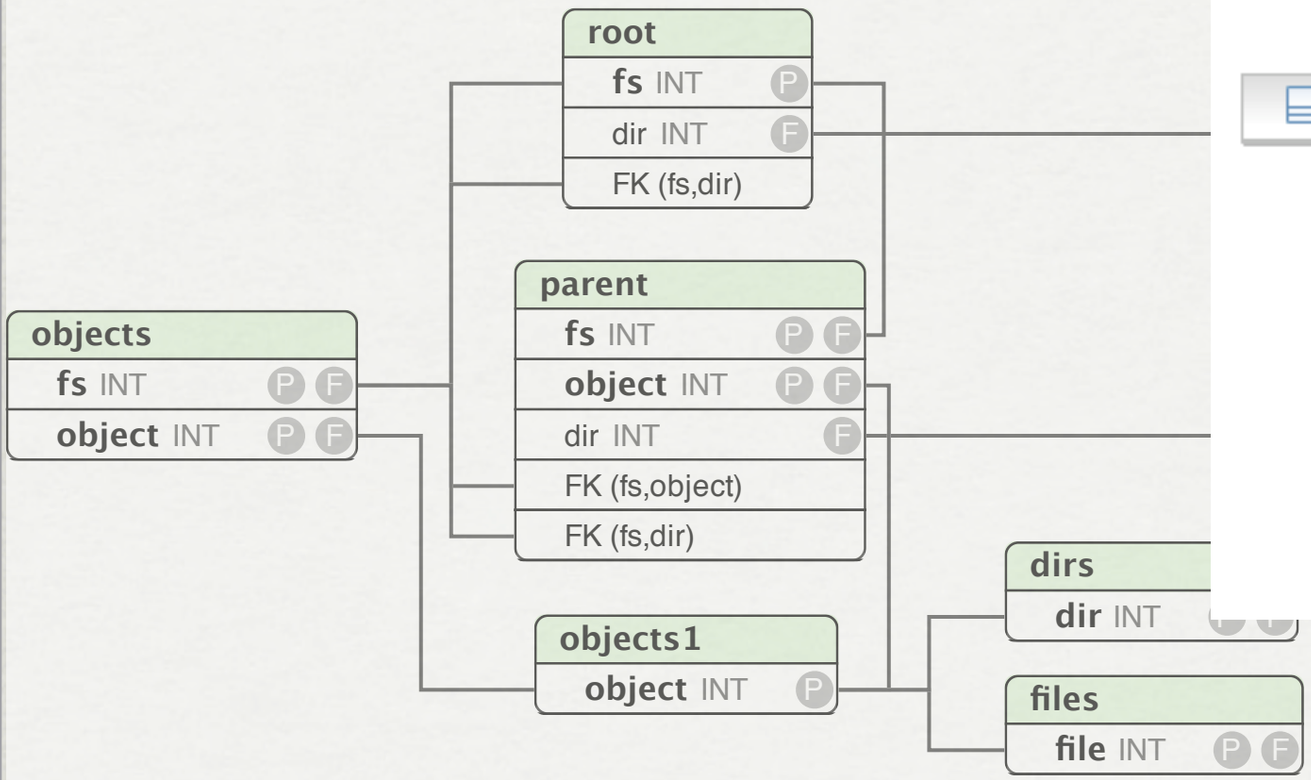
Future work:
Object-Relational Mapping

- Alloy is seen by many as an object modeling notation (object-oriented flavour)
- generate an application layer with an ORM solution
  - better interface for developers
  - solve the object-relational impedance mismatch
  - Alloy invariants as JML constraints

Future work:
Object-Relational Mapping

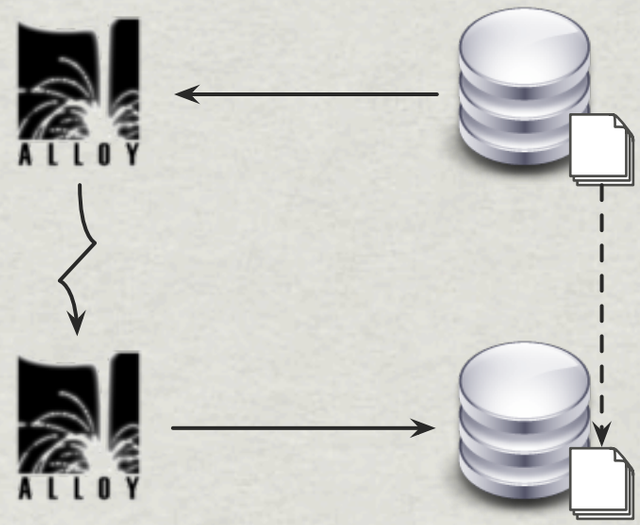
- UML class diagram for the filesystem example



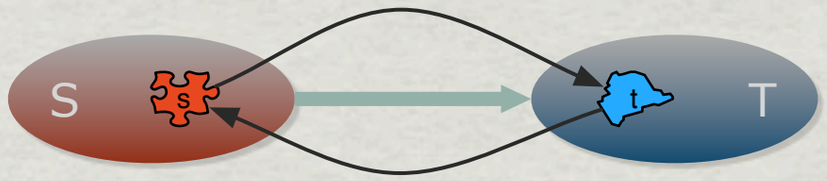


Future work:

Round-trip engineering of databases

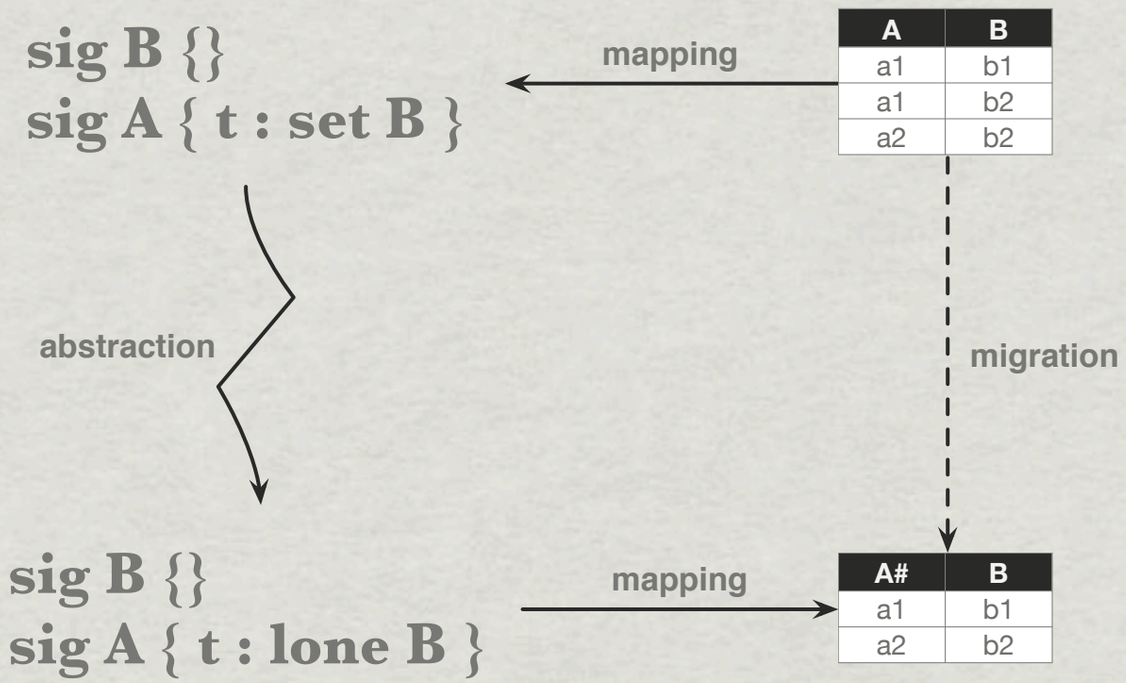


- relates to work on bidirectional two-level transformations



Future work:
Round-trip engineering of databases

- a database reengineering example



- generate a new database and a migration script