Software Engineering VU [194.020]

Maria Christakis

TU Wien

https://mariachris.github.io

Summary

Informal modeling

Strengths

- Describe particular views of the system
- Omit some information or specify it informally
- Graphical notation facilitates communication

Weaknesses

- Precise meaning of models is often unclear
- Incomplete and informal models hamper tool support
- Many details are hard to depict visually

Design

- Informal models
- Formal models
- Design by contract

Formal modeling

- Notations and tools are based on math and are precise
- Typically used to describe some aspect of a system
- Formal models enable automatic analysis
  - Finding ill-formed examples
  - Checking properties

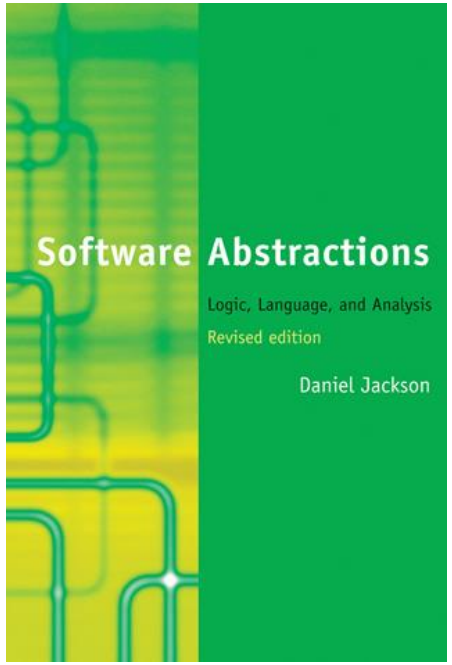
Alloy

- Alloy is a formal modeling language based on set theory
- An Alloy model specifies a collection of constraints describing a set of structures
- The Alloy analyzer reasons about the constraints of a model
  - Generates sample structures
  - Generates counterexamples for invalid properties
  - Visualizes structures

Alloy documentation and download

- https://alloytools.org

- The documentation includes
  - Tutorials
  - Book by Daniel Jackson



Design

- Informal models
- Formal models
  - Static models
  - Analyzing models
- Design by contract

Signatures

- A signature declares a set of atoms
  - Think of signatures as classes
  - Think of atoms as immutable objects
  - Different signatures declare disjoint sets

```
sig FSObject {}
```

- Extends-clauses declare subset relations
  - File and Dir are disjoint subsets of FSObject

```
sig File extends FSObject {}
sig Dir extends FSObject {}
```

Operations on sets

- Standard set operators
  - + (union)
  - & (intersection)
  - - (difference)
  - **in** (subset)
  - = (equality)
  - # (cardinality)
  - **none** (empty set)
  - **univ** (universal set)

```
sig File extends FSObject {}
sig Dir extends FSObject {}
```

```
#{f: FSObject | f in File + Dir} >= #Dir
```

```
#(File + Dir) >= #Dir
```

More on signatures

- Signatures can be abstract
  - Like abstract classes

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

  - Closed-world assumption: the declared set

contains exactly the elements of the declared subsets

```
FSObject = File + Dir
```

- Signatures may constrain the cardinalities of the declared sets
  - **one**: singleton set
  - **lone**: singleton or empty
  - **some**: non-empty set

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

Fields

- A field declares a relation on atoms
  - f is a binary relation with domain A and range given by expression e
  - Think of fields as associations

```
sig A {
  f: e
}
```

- Range expressions may denote multiplicities
  - **one**: singleton set (default)
  - **lone**: singleton or empty set
  - **some**: non-empty set
  - **set**: any set

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

Operations on relations

- Standard relation operators
  - -> (cross product)
  - . (relational join)
  - ~ (transposition)
  - ^ (transitive closure)
  - * (reflexive transitive closure)
  - <: (domain restriction)
  - >: (range restriction)
  - ++ (override)
  - **iden** (identity relation)
  - [ ] (box join: e1[e2] = e2.e1)

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

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

```
FSObject in Root.*contents
```

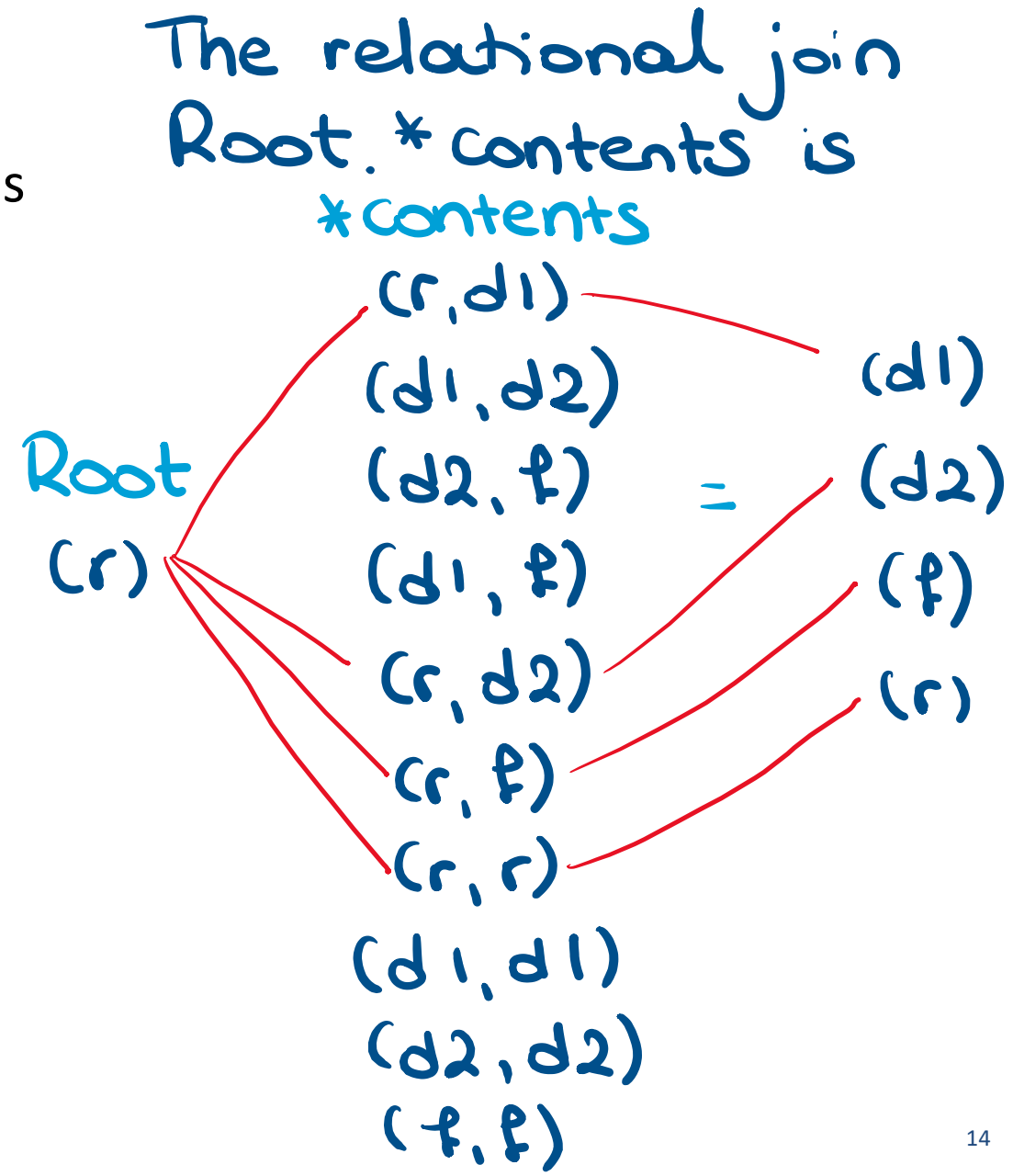
Relational join: Example

- Consider a structure with four FSObject atoms
  - r: Root, d1,d2: Dir, f: File
- And contents relation
  - (r,d1) (d1,d2) (d2,f)

- The reflexive transitive closure *contents is
  - (r,d1) (d1,d2) (d2,f)

  (d1,f) (r,d2) (r,f)

  (r,r) (d1,d1) (d2,d2) (f,f)



More on fields

- Fields may range over relations
- Relation declarations may include multiplicities on both sides
  - **one**, **lone**, **some**, **set** (default)

```
sig University {
  enrollment: Student set -> one Program
}
```

- Range expressions may depend on other fields

```
sig University {
  students: set Student,
  enrollment: students set -> one Program
}
```

Constraints

- Boolean operators
  - ! or **not** (negation)
  - && or **and** (conjunction)
  - || or **or** (disjunction)
  - => or **implies** (implication)
  - **else** (alternative)
  - <=> or **iff** (equivalence)
- Four equivalent constraints

```
F => G else H
F implies G else H
(F && G) || ((!F) && H)
(F and G) or ((not F) and H)
```

- Quantified expressions
  - **some** e

  e has at least one tuple
  - **no** e

  e has no tuples
  - **lone** e

  e has at most one tuple
  - **one** e

  e has exactly one tuple

```
no Root.parent
```

Quantification

- Alloy supports five quantifiers
  - **all** x: e | F

  F holds for every x in e
  - **some** x: e | F

  F holds for at least one x in e
  - **no** x: e | F

  F holds for no x in e
  - **lone** x: e | F

  F holds for at most one x in e
  - **one** x: e | F

  F holds for exactly one x in e

- Quantifiers may have the following forms:
  - **all** x: e | F
  - **all** x: e1, y: e2 | F
  - **all** x, y: e | F
  - **all disj** x, y: e | F

- E.g., the contents relation is acyclic

```
no d: Dir | d in d.^contents
```

Predicates and functions

- Predicates are named, parameterized formulas

```
pred p[x1:e1,…,xn:en]{F}
```

```
pred isLeaf[f:FSObject] {
  f in File || no f.contents
}
```

returns bool

- Functions are named, parameterized expressions

```
fun f[x1:e1,…,xn:en]: e {E}
```

returns e

```
fun leaves[f:FSObject]: set FSObject {
  {x: f.*contents | isLeaf[x]}
}
```

Exploring the model

- The Alloy analyzer can search for structures that satisfy the constraints M in a model

- Find instance of a predicate
  - A solution to M && **some** x1:e1,…,xn:en | F

- Find instance of a function
  - A solution to M && **some** x1:e1,…,xn:en, res:e | res = E

```
pred p[x1:e1,…,xn:en]{F}
run p
fun f[x1:e1,…,xn:en]: e {E}
run f
```

Exploring the model: Scopes

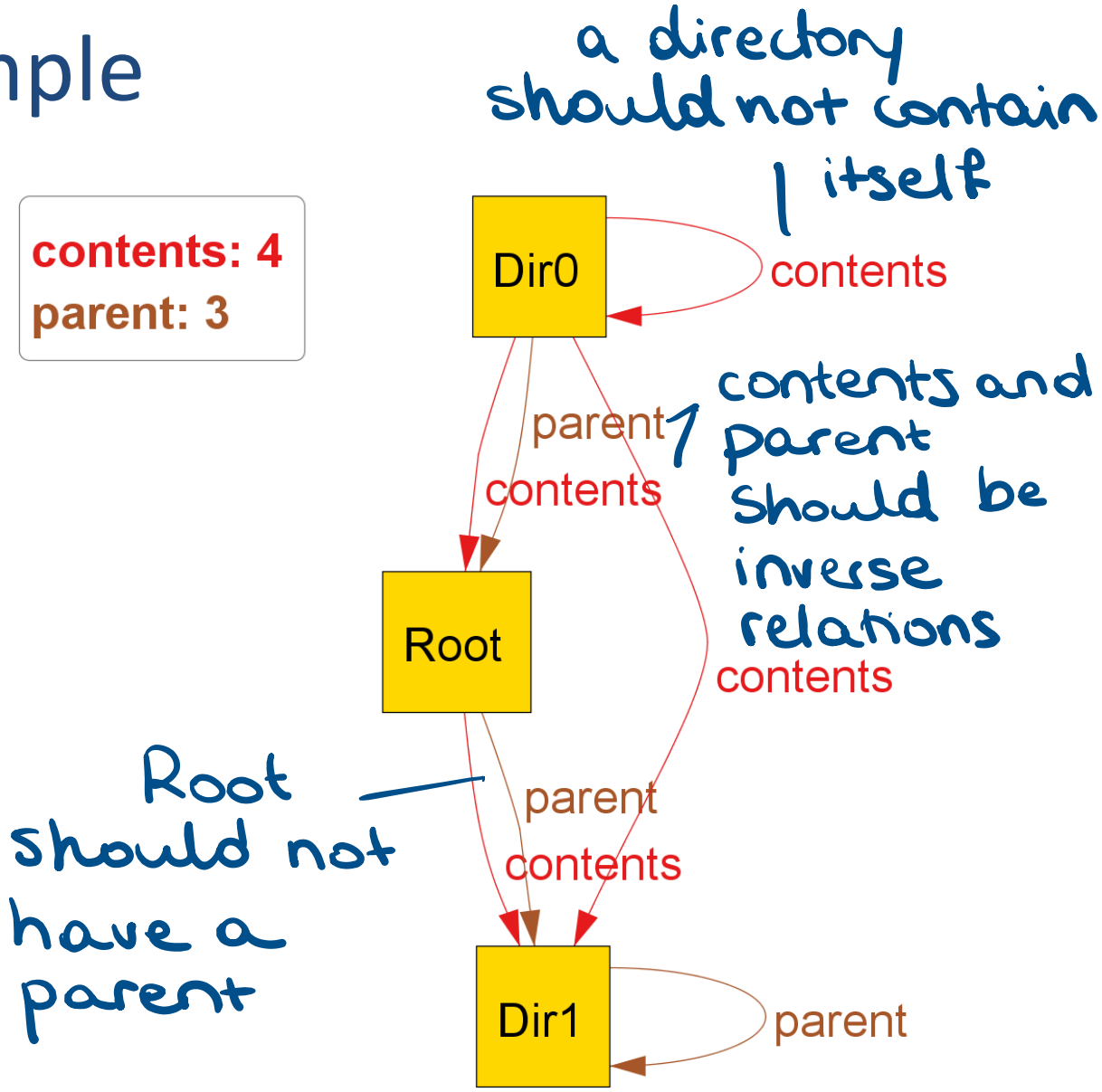
- The existence of a structure that satisfies the constraints in a model is in general undecidable

- The Alloy analyzer searches exhaustively for structures up to a given size
  - The problem becomes finite, and thus, decidable

```
run isLeaf
run isLeaf for 5
run isLeaf for 5 Dir, 2 File
run isLeaf for exactly 5 Dir
run isLeaf for 5 but 3 Dir
run isLeaf for 5 but exactly 3 Dir
```

Exploring the model: Example

```
abstract sig FSObject {
  parent: lone Dir
}

sig File extends FSObject {}

sig Dir extends FSObject {
  contents: set FSObject
}

one sig Root extends Dir {}
```



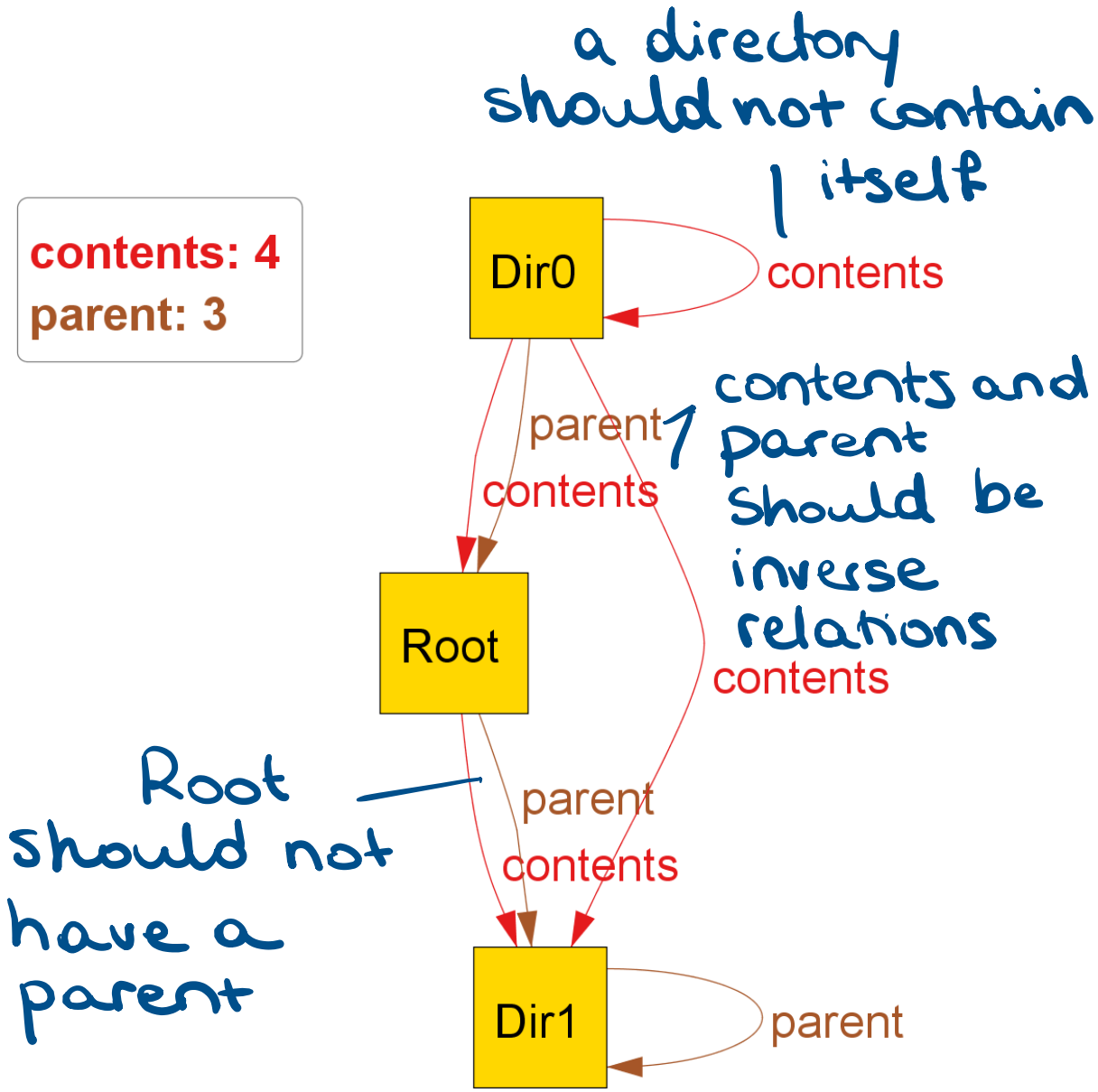
Adding constraints

- Facts add constraints that always hold
  - **run** searches for solutions that satisfy all constraints

```
fact { F }
fact f { F }
sig S {...}{F}
```

- Facts express value and structural invariants of the model

Adding constraints: Example




```
// The contents path is acyclic
fact { no d: Dir | d in d.^contents }
```

```
// A directory is the parent of its contents
fact { all d: Dir, o: d.contents | o.parent = d }
```

```
// Root is the root
fact { no Root.parent }
```

Checking the model

- Exploring models by manually inspecting instances is cumbersome
- The Alloy analyzer can search for structures that violate a given property
  - It can find counterexamples to an assertion
  - The search is complete for a given scope
- For a model with constraints M, find a solution to M && !F

```
assert a { F }
```

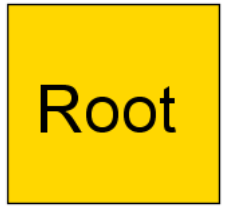
```
check a scope
```

Checking the model: Example

- Finding a counterexample

```
pred isLeaf[f: FSObject] {
  f in File || no f.contents
}
```

```
assert nonEmptyRoot { !isLeaf[Root] }
check nonEmptyRoot for 3
```



Validity is checked only within the given scope

- Proving a property

```
assert acyclic { no d: Dir | d in d.^contents }
check acyclic for 5
```

```
Executing "Check acyclic for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1212 vars. 63 primary vars. 2423 clauses. 62ms.
   No counterexample found. Assertion may be valid. 0ms.
```

Under and over-constrained models

- Missing or weak facts under-constrain the model
  - They permit undesired structures
  - Under-constrained models are typically easy to detect during model exploration (using **run**) and assertion checking (using **check**)

- Unnecessary facts over-constrain the model
  - They exclude desired structures

- Inconsistencies are an extreme case of over-constraining
  - They preclude the existence of any structure
  - All assertion checks will succeed!

```
// The contents path is acyclic
fact acyclic { no d: Dir | d in d.*contents }

assert nonsense { 0 = 1}
check nonsense
```

Guidelines to avoid over-constraining

- Simulate model to check consistency
  - Use **run** to ensure that structures exist

```
// The contents path is acyclic
fact acyclic { no d: Dir | d in d.*contents }

pred show {}
run show
```

```
Executing "Run show"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=4 Symmetry=20
   0 vars. 0 primary vars. 0 clauses. 16ms.
   No instance found. Predicate may be inconsistent. 0ms.
```

- Prefer assertions over facts
  - When in doubt, check whether the current model already ensures a desired property before adding it as a fact

Design

- Informal models
- Formal models
  - Static models
  - Analyzing models
- Design by contract

Consistency and validity

- An Alloy model specifies a collection of constraints C that describe a set of structures

- Consistency:

A formula F is consistent (satisfiable) if it evaluates to true in at least one of these structures

$$\exists s \bullet C(s) \wedge F(s)$$

- Validity:

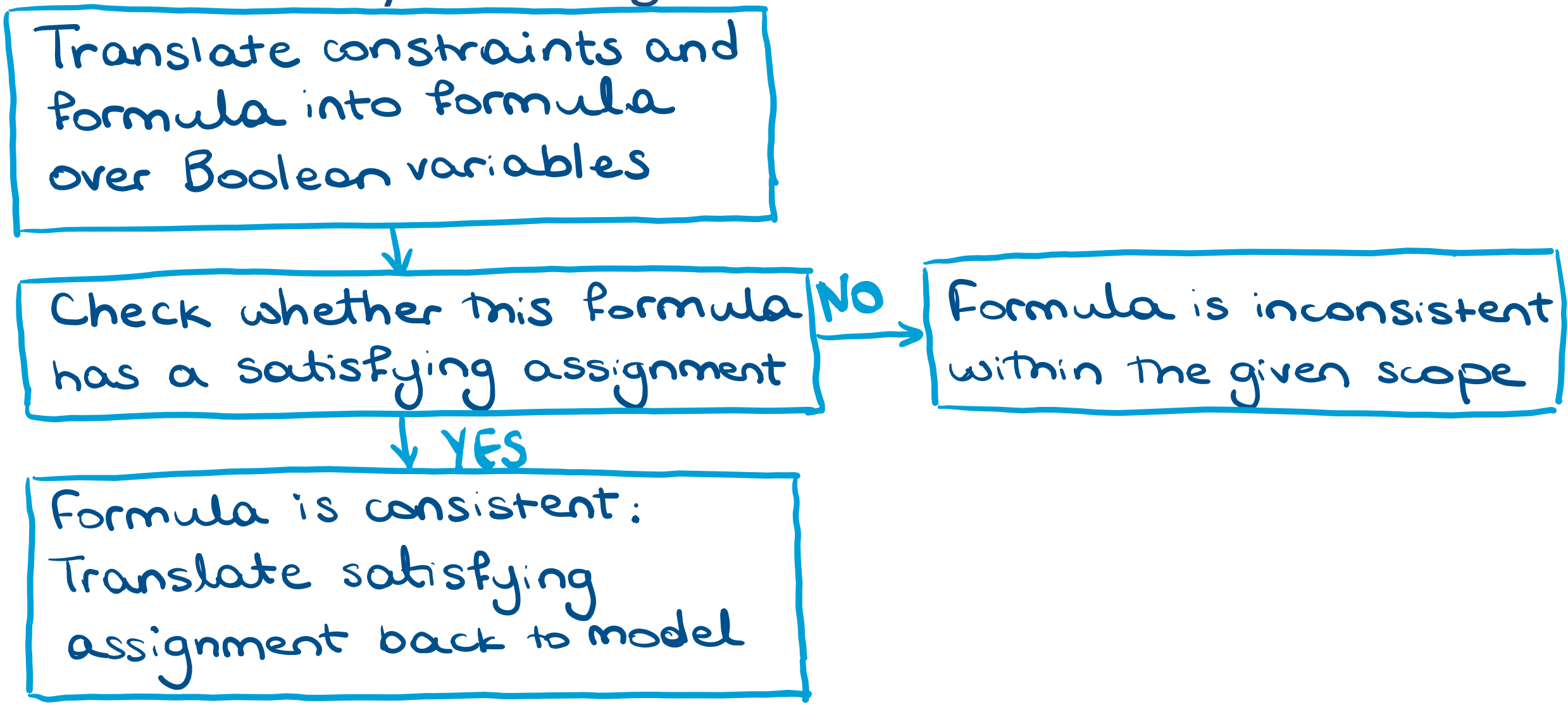
A formula F is valid if it evaluates to true in all these structures

$$\forall s \bullet C(s) \Rightarrow F(s)$$

Analyzing models within a scope

- Validity and consistency checking for Alloy is undecidable
- The Alloy analyzer sidesteps this problem by checking validity and consistency within a given scope
  - A scope gives a finite bound on the sizes of the sets in the model (which makes everything else in the model also finite)
  - Naïve algorithm: enumerate all structures of a model within the bounds and check the formula for each of them

Consistency checking



Translation into Boolean formula

- Internally, Alloy represents all data types as relations
  - A relation is a set of tuples

```
sig Node {
  next: lone Node
}
```

next is a binary relation in Node x Node

- Constraints and formulas in the model are represented as formulas over relations

```
fact {
  all n: Node | n != n.next
}
```

$\forall n \bullet (n, n) \notin next$

Translation into Boolean formula

- A relation is translated into Boolean variables
  - Introduce one Boolean variable for each tuple that is potentially contained in the relation

```
sig Node {
  next: lone Node
}
pred show {}
run show for 3
```

next is a binary relation in Node x Node

$n_{00}, n_{01}, n_{02}, n_{10}, n_{11}, n_{12}, n_{20}, n_{21}, n_{22}$

For the given scope, next may contain 9 different tuples

- Constraints and formulas are translated into Boolean formulas over these variables

```
fact {
  all n: Node | n != n.next
}
```

$\neg(n_{00} \wedge n_{01}) \wedge \neg(n_{00} \wedge n_{02}) \wedge \neg(n_{01} \wedge n_{02}) \wedge$
$\neg(n_{10} \wedge n_{11}) \wedge \neg(n_{10} \wedge n_{12}) \wedge \neg(n_{11} \wedge n_{12}) \wedge$
$\neg(n_{20} \wedge n_{21}) \wedge \neg(n_{20} \wedge n_{22}) \wedge \neg(n_{21} \wedge n_{22}) \wedge$
$\neg n_{00} \wedge \neg n_{11} \wedge \neg n_{22}$

Check for satisfying assignments

- Satisfiability of formulas over Boolean variables is a well understood problem
  - Find a satisfying assignment if one exists and return UNSAT otherwise
  - The problem is NP-complete
- In practice, SAT solvers are extremely efficient

A satisfying assignment

$$\neg(n_{00} \wedge n_{01}) \wedge \neg(n_{00} \wedge n_{02}) \wedge \neg(n_{01} \wedge n_{02}) \wedge$$
$$\neg(n_{10} \wedge n_{11}) \wedge \neg(n_{10} \wedge n_{12}) \wedge \neg(n_{11} \wedge n_{12}) \wedge$$
$$\neg(n_{20} \wedge n_{21}) \wedge \neg(n_{20} \wedge n_{22}) \wedge \neg(n_{21} \wedge n_{22}) \wedge$$
$$n_{00} \wedge n_{11} \wedge n_{22}$$

n	0	1	2
0	F	F	F
1	F	F	T
2	F	T	F

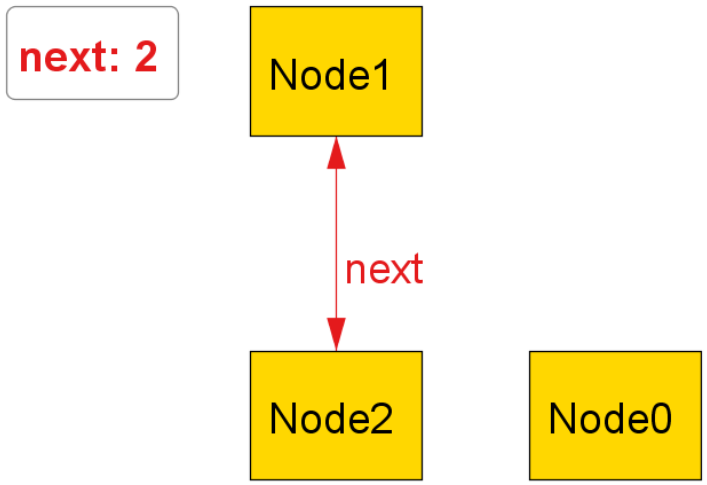
Translation back to model

- A satisfying assignment can be translated back to relations

n	0	1	2
0	F	F	F
1	F	F	T
2	F	T	F

next = {(1,2), (2,1)}

and then visualized



Interpretation of UNSAT

- If a Boolean formula has no satisfying assignment, the SAT solver returns UNSAT

- The Boolean formula encodes an Alloy model within a given scope
  - There are no structures within this scope, but larger structures may exist
  - The model may be, but is not necessarily inconsistent

```
sig Node {
  next: lone Node
}
fact { #Node = 4 }
pred show {}
run show for 3
```

```
Executing "Run show for 3"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  0 vars. 0 primary vars. 0 clauses. 0ms.
  No instance found. Predicate may be inconsistent. 0ms.
```

Validity and invalidity checking

- A formula F is valid if it evaluates to true in all structures that satisfy the constraints C of the model

$$\forall s . C(s) \Rightarrow F(s)$$

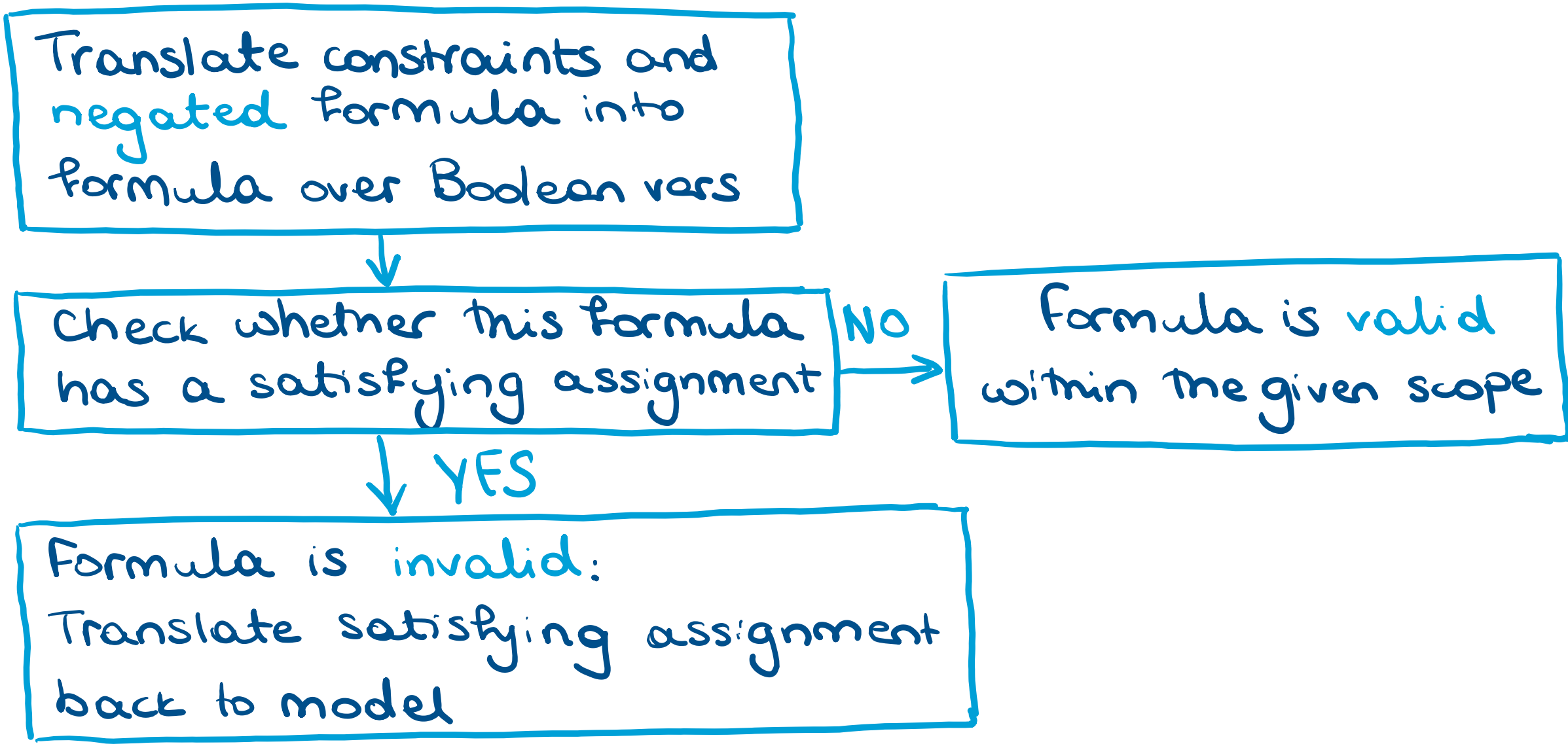
- Enumerating all structures within a given scope is possible but too slow

- Instead of checking validity, the Alloy analyzer checks for invalidity, that is, it looks for counterexamples

$$\neg(\forall s . C(s) \Rightarrow F(s)) \equiv \exists s . C(s) \wedge \neg F(s)$$

this is a consistency check

Validity checking



Interpretation of UNSAT

- Validity checking searches for a counterexample within a given scope
  - There are no structures within this scope, but larger structures may exist
  - The model may be, but is not necessarily valid

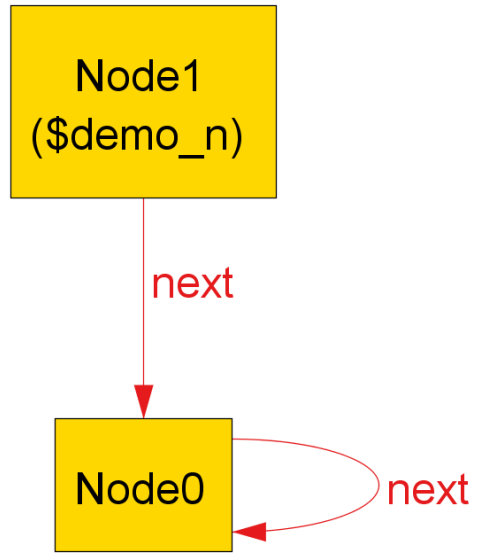
```
sig Node { next: Node }

assert demo { all n: Node | some m: Node | m.next = n }
```

```
check demo for 1
```

```
Executing "Check demo for 1"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  14 vars. 3 primary vars. 18 clauses. 9ms.
  No counterexample found. Assertion may be valid. 0ms.
```

```
check demo for 2
```

next: 2



Analyzing models: Summary

- Consistency checking
  - Performed by **run** command within a scope
  - Positive answers are definite (structures)

- Validity checking
  - Performed by **check** command within a scope
  - Negative answers are definite (counterexamples)

- Small model hypothesis

Most interesting errors are found by looking at small instances

Design

- Informal models
- Formal models
  - Static models
  - Analyzing models
- Design by contract

Design by contract

- Pioneered by Bertrand Meyer in the Eiffel language
- Defining formal, precise, and verifiable interface specifications for software components, with preconditions, postconditions, and invariants
- Three key questions that the designer must repeatedly ask:
  - What does this code expect?
  - What does it guarantee?
  - What does in maintain?

Preconditions

- Preconditions express requirements on the input state (parameters, heap) of a method

- Semantics
  - Condition must be true at the entry of the method

```
// requires 0 <= index < size
public int getElemAtIndex(int index) {
  return elems[index];
}
```

Checking preconditions

- Approach 1: Return error value

```
// requires 0 <= index < size
public int getElemAtIndex(int index) {
  if (index < 0 || index >= size) {
    return -1;
  } else {
    return elems[index];
  }
}
```

Checking preconditions

- Approach 2: Throw exception

```
// requires 0 <= index < size
public int getElemAtIndex(int index) {
  if (index < 0 || index >= size) {
    throw new IndexOutOfBoundsException(index);
  } else {
    return elems[index];
  }
}
```

Checking preconditions

- Approach 3: Use assertions
  - An assertion is a logical statement that can be made at a particular program point and is expected to be true
  - In Java, assertions can be enabled (they are off by default)

```
// requires 0 <= index < size
public int getElemAtIndex(int index) {
  assert (0 <= index && index < size) : “index must be within bounds”;
  return elems[index];
}
```

Postconditions

- Postconditions express guarantees about the result and output state (out-parameters, heap) of a method

- Semantics
  - Condition must be true at the normal exit of the method

```
// requires 0 <= capacity
// ensures capacity <= elems.length
public int ensureCapacity(int capacity) {
  while (capacity > elems.length) {
    elems = Arrays.copyOf(elems, 2*elems.length);
  }
}
```

Checking postconditions

- Use assertions

```
// requires 0 <= capacity
// ensures capacity <= elems.length
public int ensureCapacity(int capacity) {
  while (capacity > elems.length) {
    elems = Arrays.copyOf(elems, 2*elems.length);
  }
  assert (capacity <= elems.length) : “elems does not have enough capacity”;
}
```

Pre- and postconditions: Methods and callers

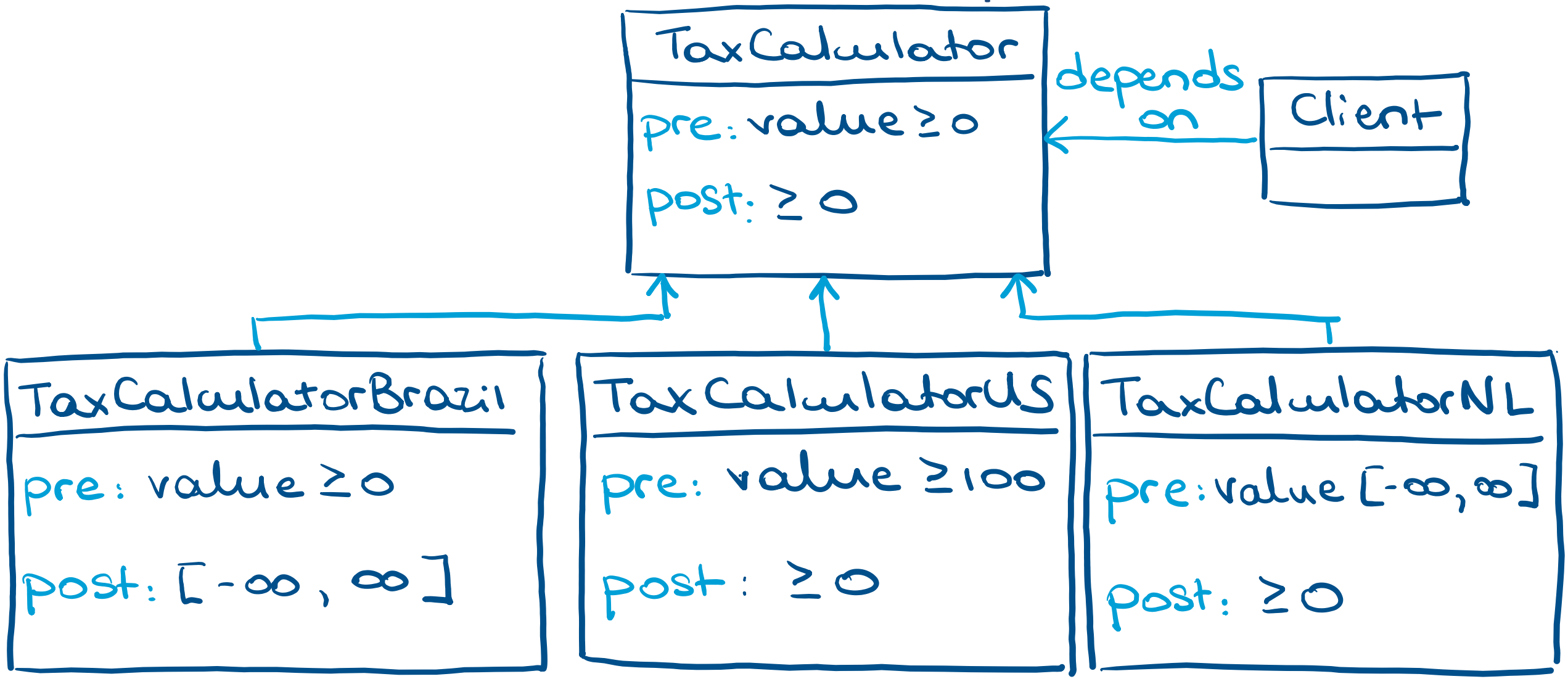
	Obligation	Benefit
Method	Establish post-condition for all calls that satisfy precondition	May assume precondition upon method entry
Caller	Establish pre-condition of called method	May assume postcondition upon return from call

Dafny

- There are many languages that natively support design by contact
- Dafny is a such a programming language that comes with a program verifier for automatically checking the specifications
- Dafny compiles to mainstream languages, such as Java, C#, Go, etc.
- Check it out: https://dafny.org/
  - Tutorials, examples, documentation, …

Dafny

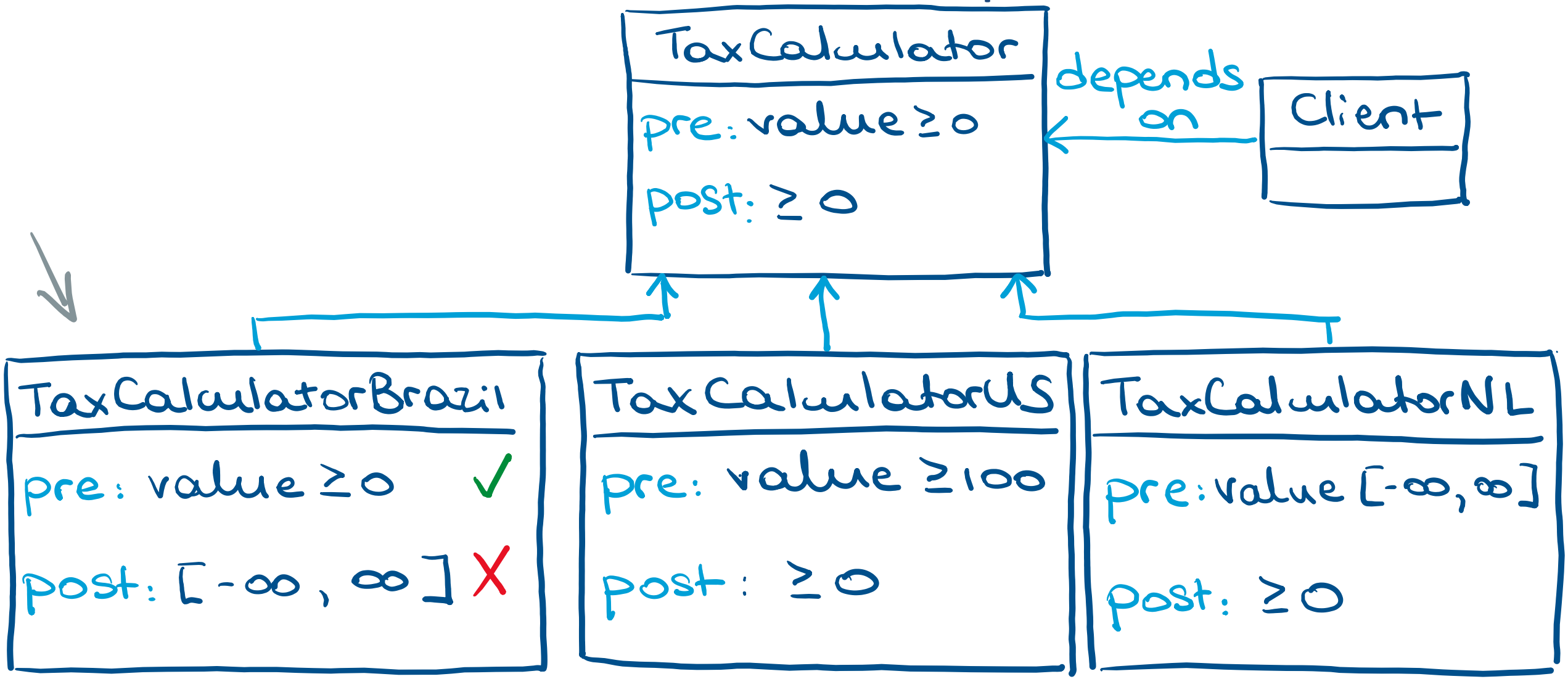
Inheritance and contracts: Example



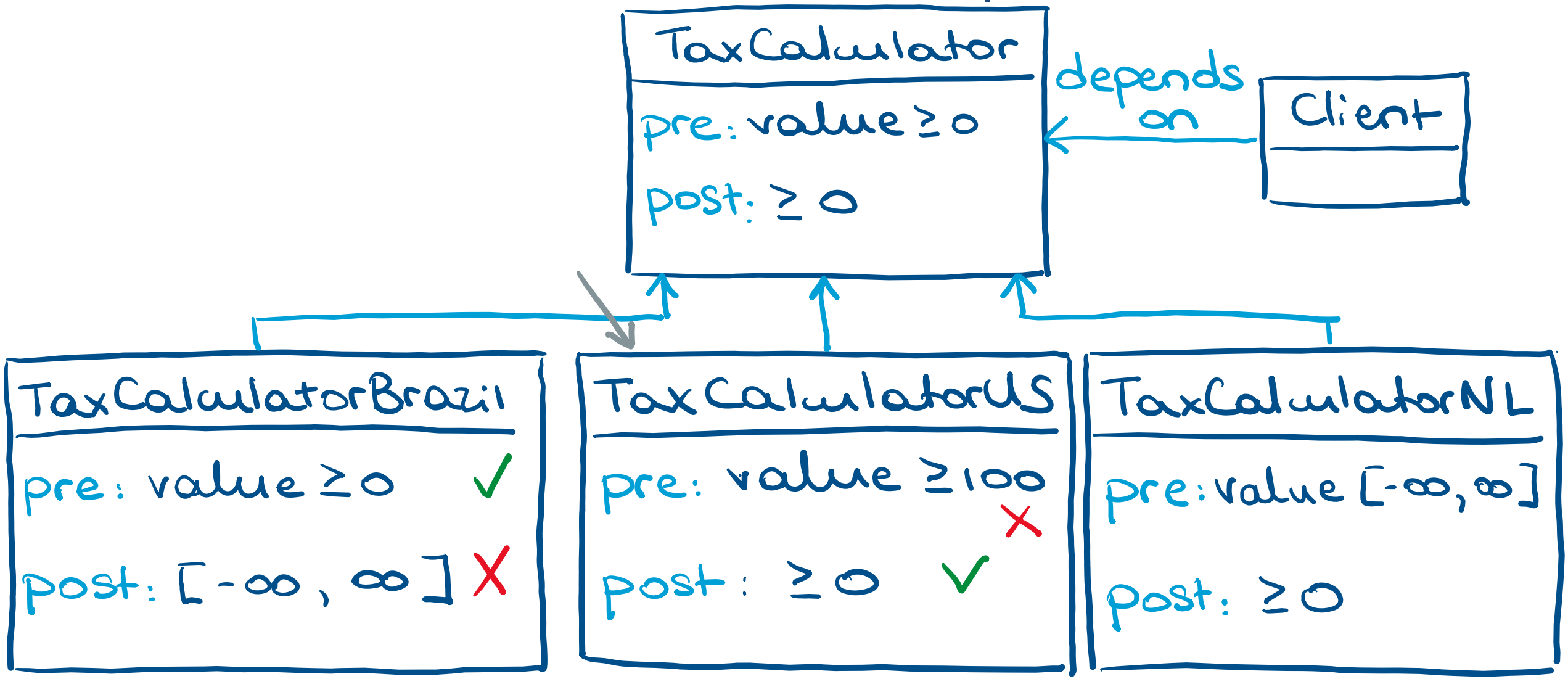
Inheritance and contracts: Example

- Suppose the client class receives a TaxCalculator in its constructor and uses it in its methods
- Due to polymorphism, any of the child classes can also be passed to the client
- Since the client does not know which tax calculator was given to it, it can only assume that whatever class it received will respect the pre- and post-conditions of the base class (the only class the client knows)

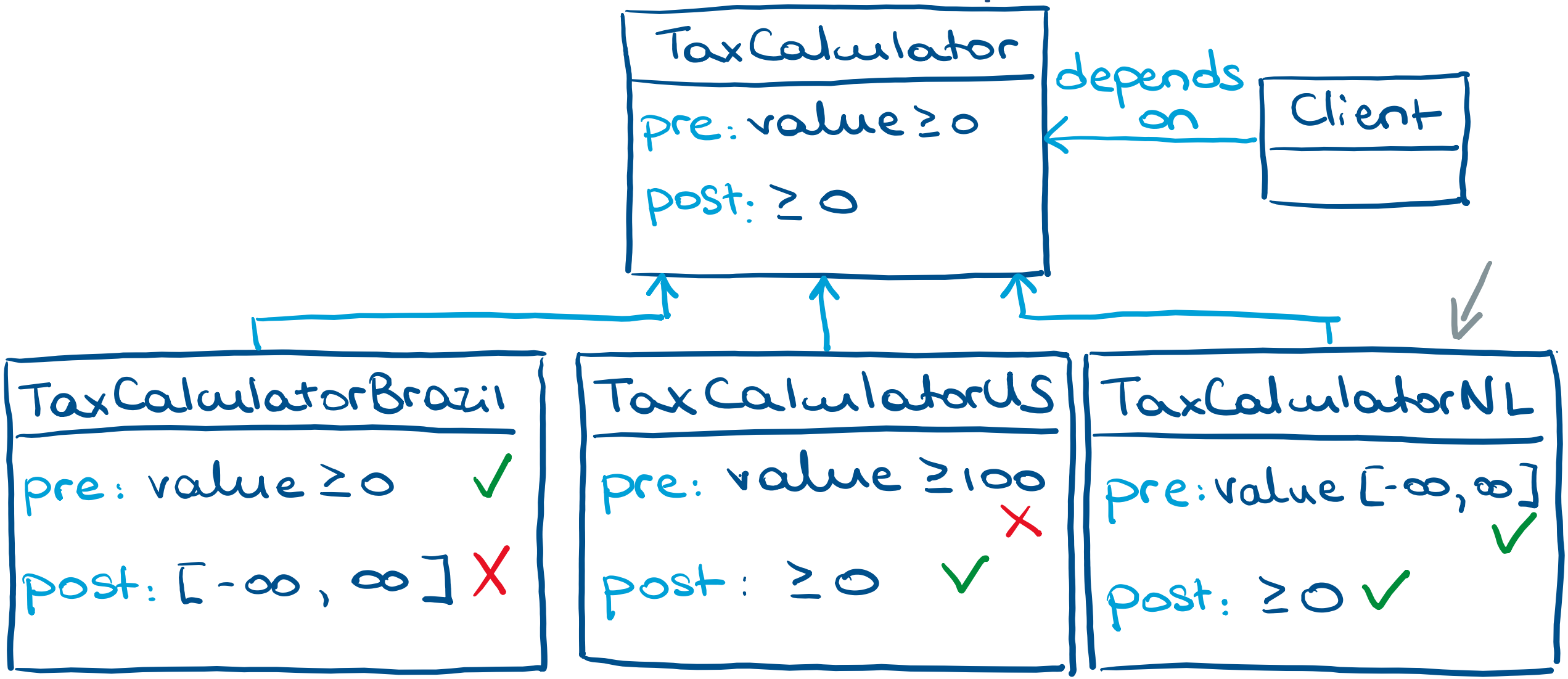
Inheritance and contracts: Example



Inheritance and contracts: Example



Inheritance and contracts: Example



Inheritance and contracts: Rules

Whenever a subclass S (e.g., TaxCalculatorBrazil) inherits from a base class B (e.g., TaxCalculator):

1. The pre-conditions of subclass S should be the same as or weaker (accept more values) than the pre-conditions of base class B

2. The post-conditions of subclass S should be the same as or stronger (return fewer values) than the post-conditions of base class B

Liskov substitution principle (LSP): The idea that a subclass may be used as a substitution for a base class without breaking the system's expected behavior

Weak or strong pre-conditions?

- An important design decision when modeling contracts is whether to use strong or weak contracts: it's a trade-off!

- Weak pre-condition
  - E.g., the method accepts any input value, including null
  - It's easy to use for clients: any call to it will work and the method will never throw an exception related to a pre-condition being violated
  - This puts an extra burden on the method, as it must handle any invalid inputs

Weak or strong pre-conditions?

- An important design decision when modeling contracts is whether to use strong or weak contracts: it's a trade-off!

- Strong pre-condition
  - E.g., the method only accepts positive numbers and does not accept null
  - This puts an extra burden on the client, as it must make sure it does not violate the pre-conditions of the method
  - The method implementation is easier, as it may assume that inputs are always valid

Suggested reading

From Book: "Effective Software Testing A Developer's Guide" by M. Aniche

- Section 4.3.1
- Section 4.5.1