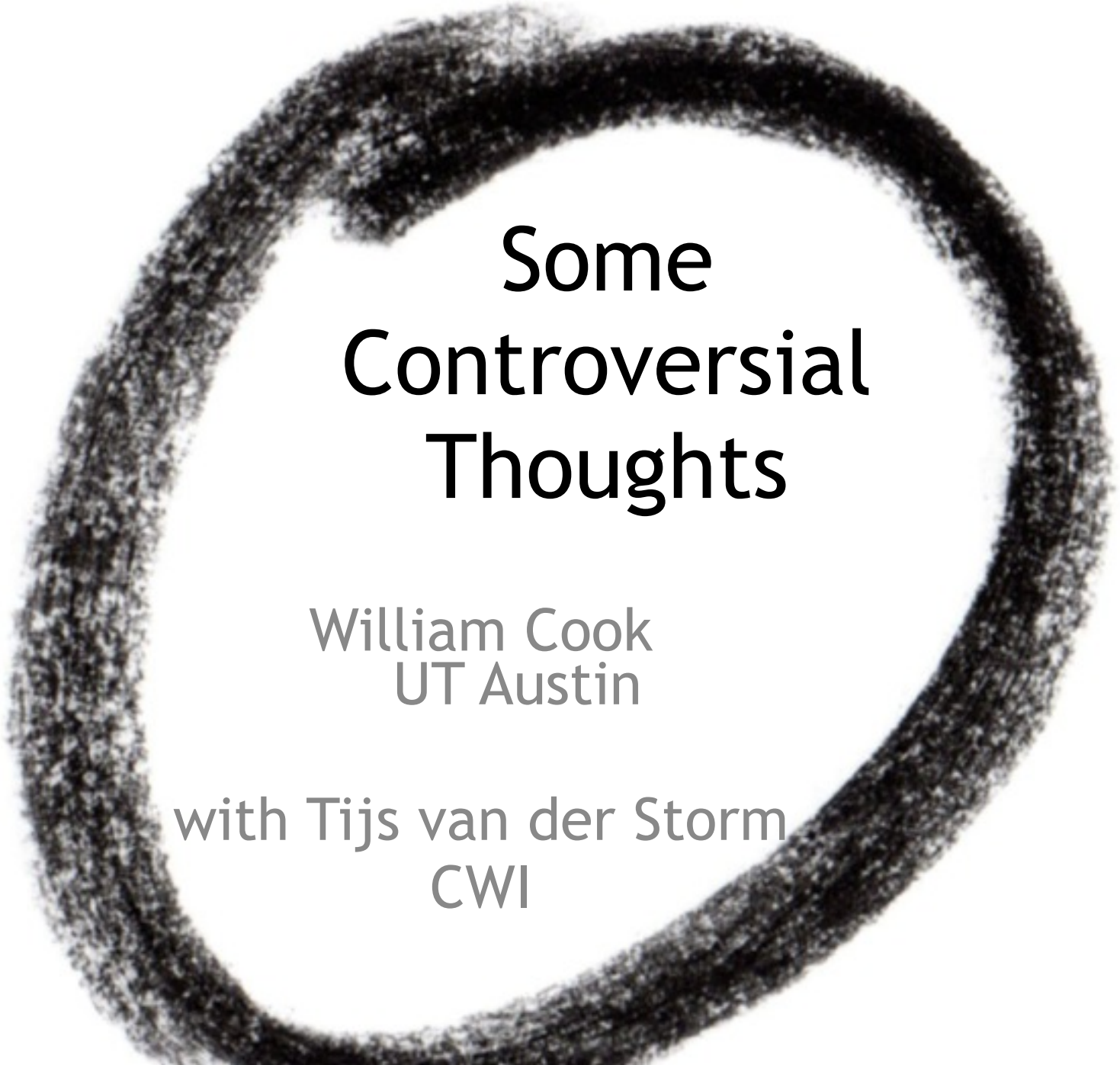


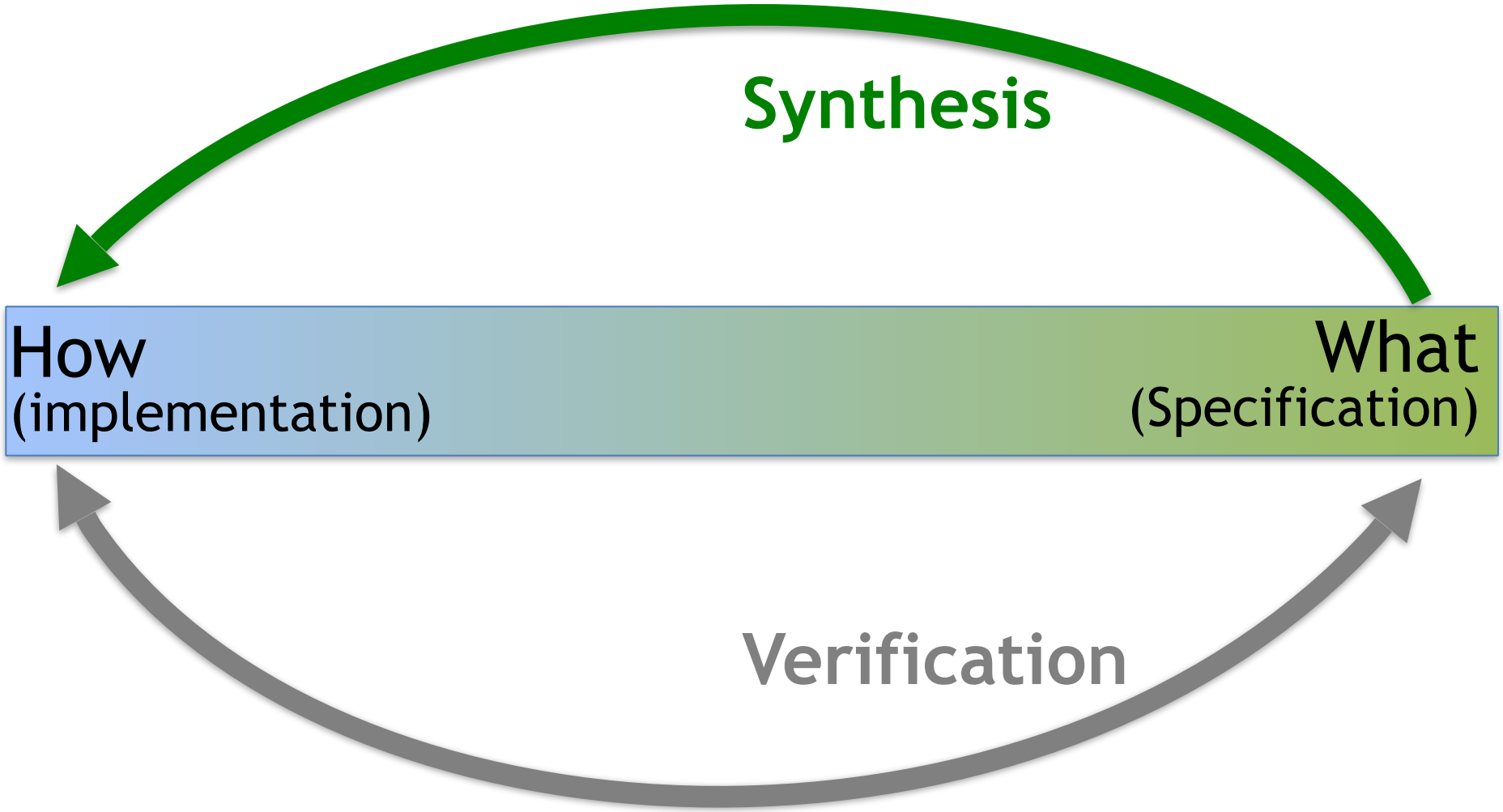
Some Controversial Thoughts

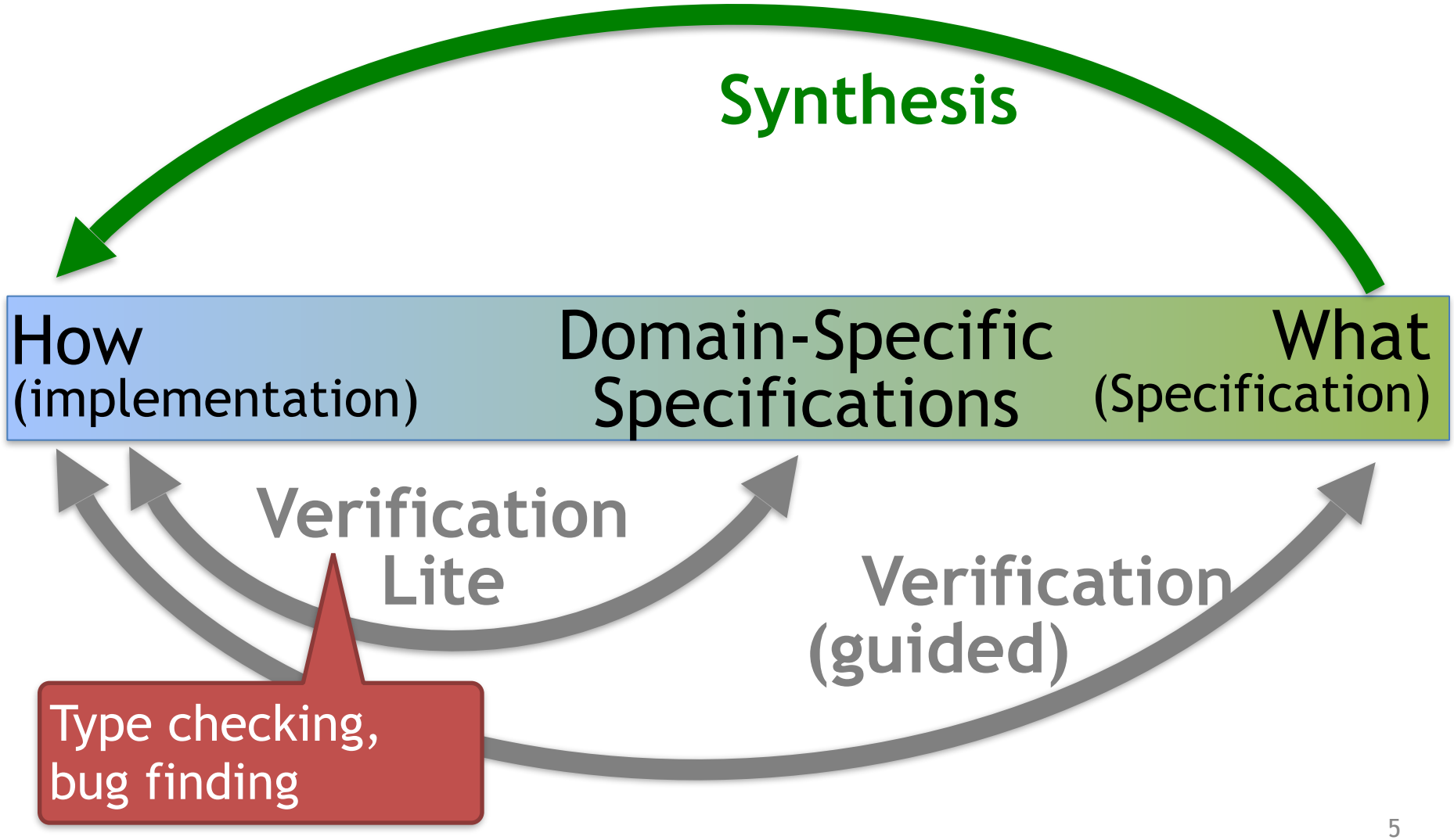
William Cook
UT Austin

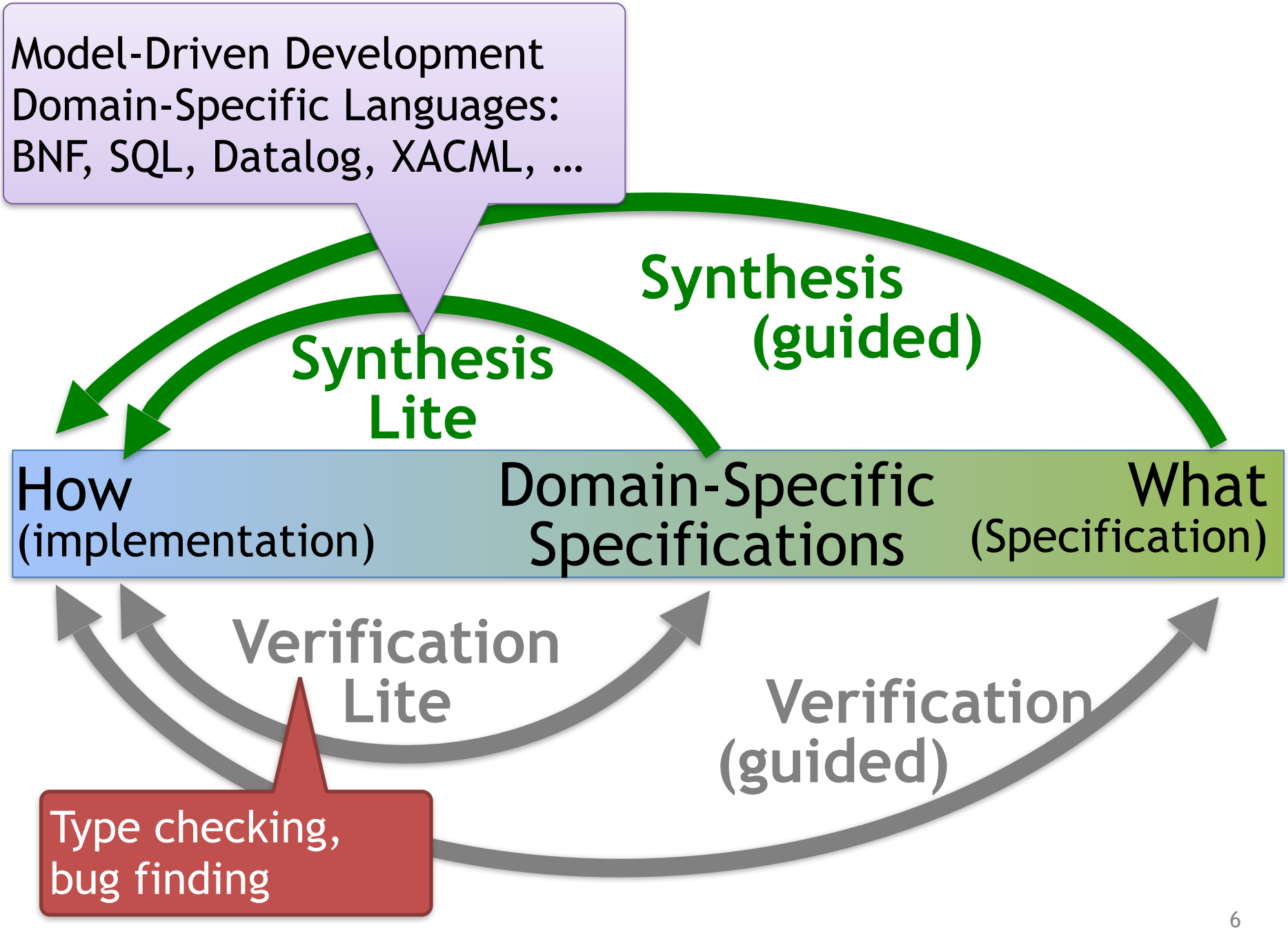
with Tijs van der Storm
CWI

Spectrum of programming


How
(implementation)
What
(Specification)
Verification


Synthesis
How
(implementation)
What
(Specification)
Verification


Synthesis
How
(implementation)
Domain-Specific
Specifications
What
(Specification)
Verification
Lite
Verification
(guided)
Type checking,
bug finding


Model-Driven Development
Domain-Specific Languages:
BNF, SQL, Datalog, XACML, …
Synthesis
(guided)
Synthesis
Lite
How
(implementation)
Domain-Specific
Specifications
What
(Specification)
Verification
Lite
Verification
(guided)
Type checking,
bug finding

STOP

working on verification

Work on Synthesis instead!

Prevent Bad

Enable Good

Bug Finding
Race Detection
Type Checking
etc.

Prevent Bad

Enable Good

Bug Finding
Race Detection
Type Checking
etc.

Prevent Bad

Enable Good

New languages?
New features?
For what?

Bug Finding
Race Detection
Type Checking
etc.

Prevent Bad

Advantages:
Measurable,
Domain-free

Enable Good

New languages?
New features?
For what?

A Problem

1. Many (many!) repeated instances of *similar* code

2. Unique *details* and *names* prevent generalization

Requirements (what)

Strategies (how)

Application (Code)

Behavior

Requirements (what)

***Small change to** Strategies*

Very different Code!!!

Behavior

Requirements (what)

Small change to *Strategies*

Chaos!

Very different Code

Behavior

Requirements (what)

Reify!

Strategies (how)

Application (Code)

Behavior

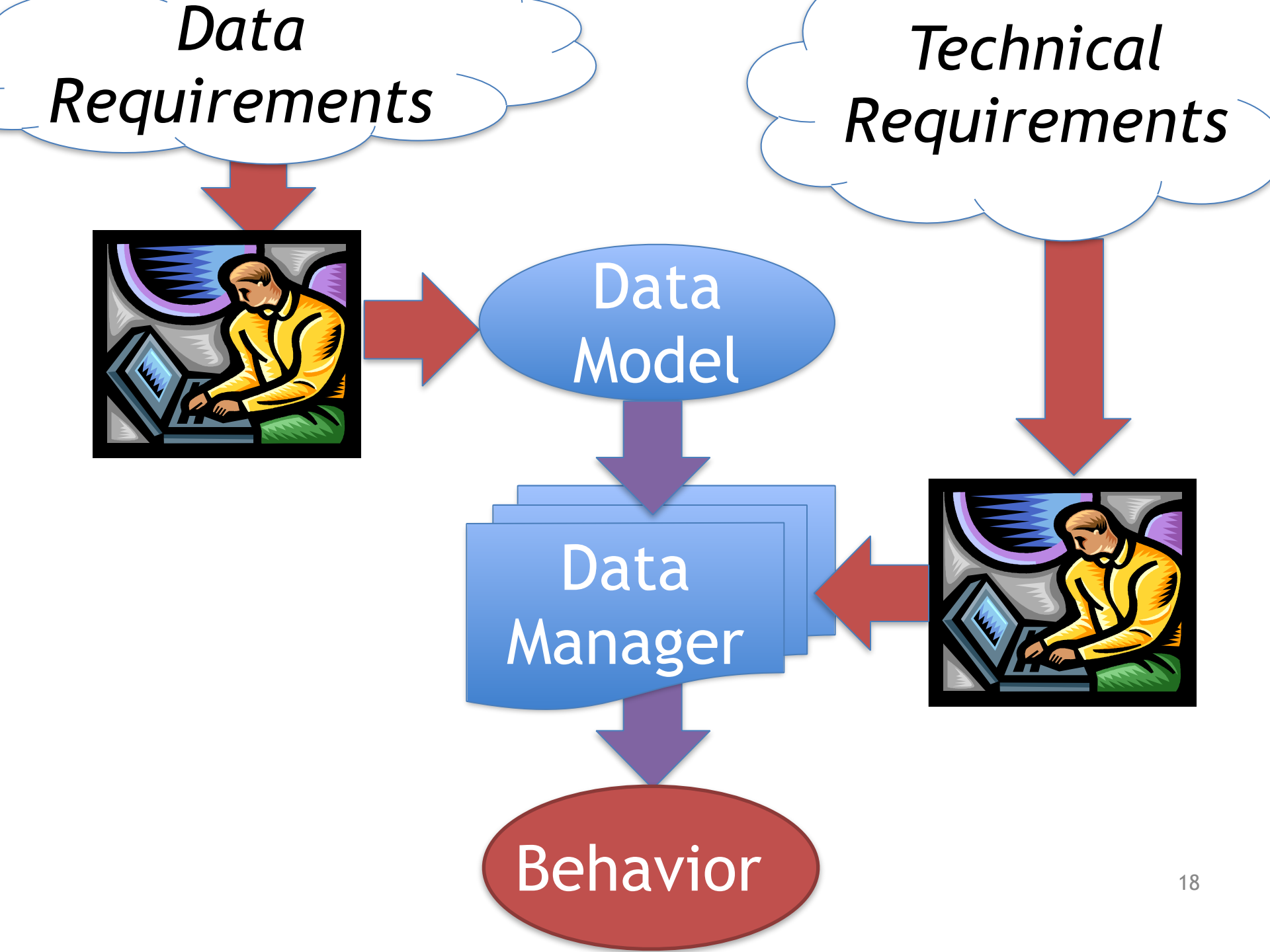
Requirements

Technical Requirements

Problem Specific

General Strategies

Behavior


Data Requirements
Technical Requirements
Data Model
Data Manager
Behavior

Using Managed Data (Ruby)

- Description of data to be managed

```
Point = { x: Integer, y: Integer }
```

- Dynamic creation based on metadata

```
p = BasicRecord.new Point
p.x = 3
p.y = -10
print p.x + p.y
p.z = 3  # error!
```

- *Factory* BasicRecord: Description<T> → T

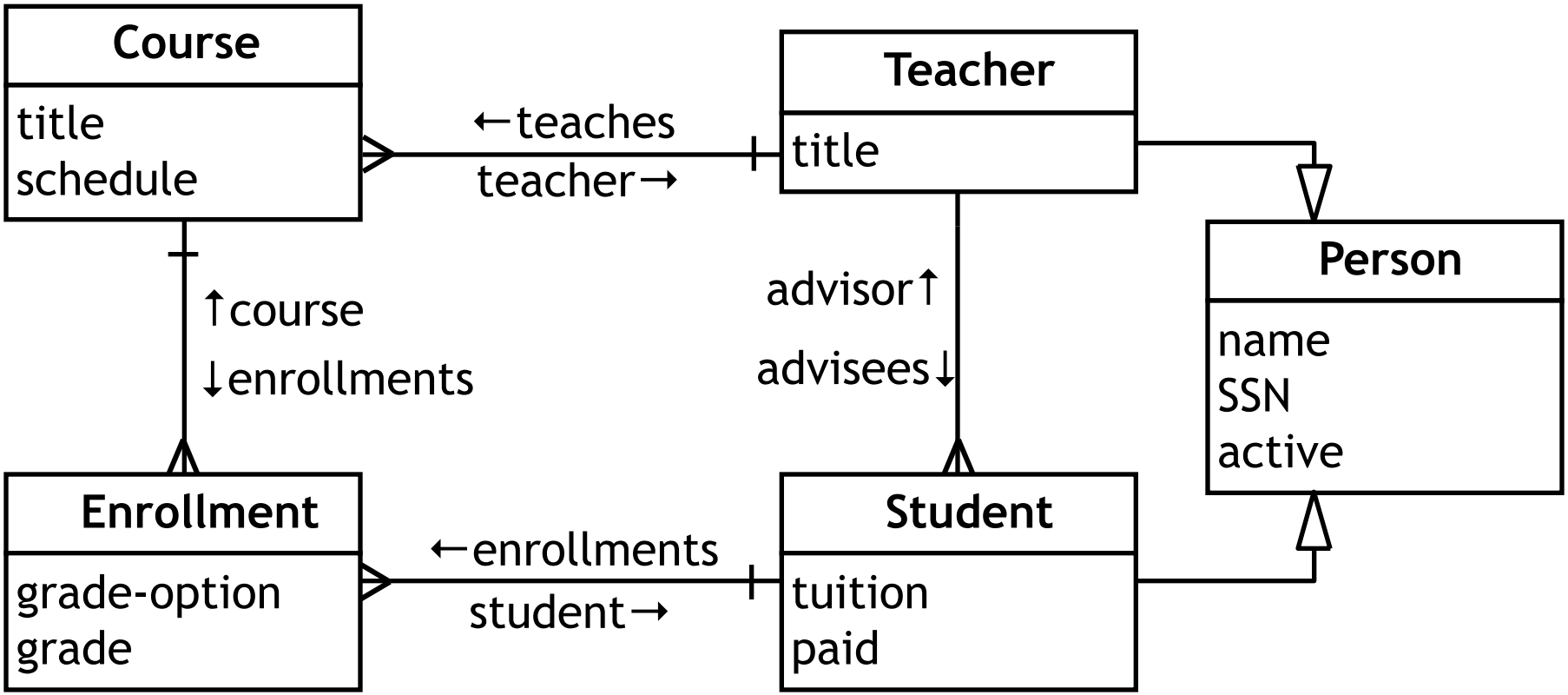
Implementing Managed Data

- Override the "dot operator" (p.x)
- Reflective handling of unknown methods
  - Ruby method_missing
  - Smalltalk: doesNotUnderstand
  - Also IDispatch, Python, Objective-C, Lua, CLOS
  - Martin Fowler calls it "Dynamic Reception"
- Programmatic method creation
  - E.g. Ruby define_method
- Partial evaluation

Other Data Managers

- Mutability: control whether changes allowed
- Observable: posts notifications
- Constrained: checks multi-field invariants
- Derived: computed fields (reactive)
- Secure: checks authorization rules
- Graph: inverse fields (bidirectional)
- Persistence: store to database/external format
- General strategy for all accesses/updates
- Combine them for *modular strategies*

Graphs, Invariants, Computed



Constraints: for all student s

s.dept = s.advisor.dept

Computed values/attribute grammars

Grammars

- Mapping between *text* and *object graph*
- A *point* is written as (x, y)

Individual	*Grammar*
(3, 4)	***P ::= [Point] "(" x:int "," y:int ")"***

class

fields

- Notes:
  - Direct reading, no abstract syntax tree (AST)
  - Bidirectional: can parse and pretty-print
  - GLL parsing, *interpreted!*

State Machine Example

Door StateMachine

```
start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed
```

StateMachine Grammar

```
M::= [Machine] "start" \start:</states[it]> states:S*
S ::= [State] "state" name:sym   out:T*
T ::= [Trans] "on" event:sym "go" to:</states[it]>
```

A StateMachine Interpreter

```
def run_state_machine(m)
  current = m.start
  while gets
    puts "#{current.name}"
    input = $_.strip
    current.out.each do |trans|
      if trans.event == input
        current = trans.to
        break
      end
    end
  end
end
```

StateMachine Schema

```
class Machine
  start  : State
  states! State*

class State
  machine: Machine
  name   #   str
  out    !   Trans*
  in     :   Trans*

class Trans
  event  :  str
  from   :  State / out
  to     :  State / in
```

Expression Example

Sample Expression

```
3*(5+6)
```

Expression Grammar

```
E   ::= [Add] left:E "+" right:M   | M
M   ::= [Mul]  left:M "*" right:P   | P
P   ::= [Num] val:int               | "(" E ")"
```

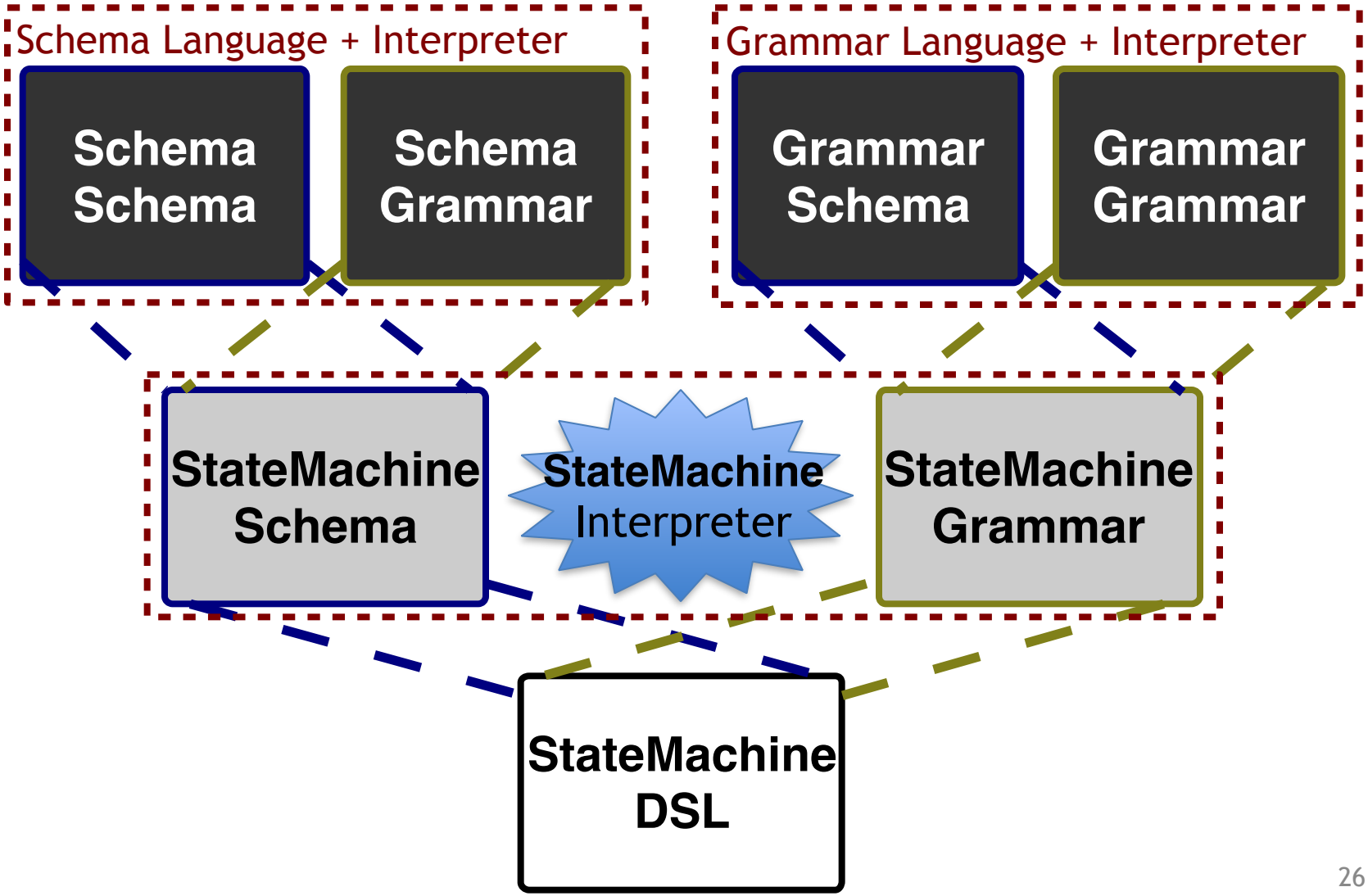
An *Expression* Interpreter

```
module Eval
  operation :eval

  def eval_Num(val)
    val
  end

  def eval_Add(left, right)
    left.eval + right.eval
  end

  def eval_Mul(left, right)
    left.eval * right.eval
  end
end
```

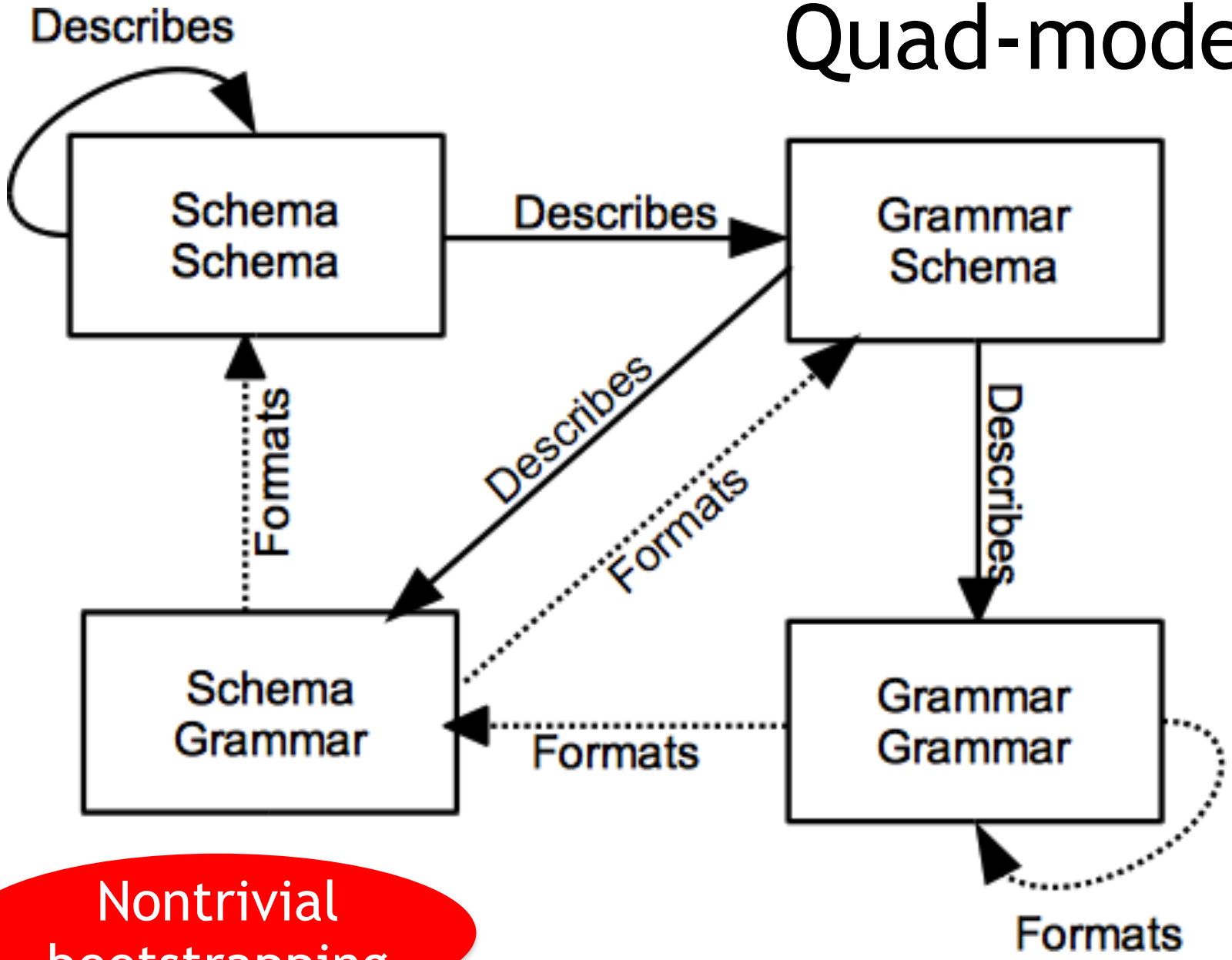
Expression Schema

```
class Exp

class Num
  val : int

class Add
  left  : Exp
  right : Exp

class Mul
  left  : Exp
  right : Exp
```

Everything is a language



Quad-model



Nontrivial bootstrapping

Schema Schema

```
class Schema
    types: Type*
class Type
    name: string
class Primitive < Type
class Class < Type
    fields: Field*
    super: Type?
```

```
class Field
    name: string
    type: Type
    many: bool
    optional: bool

primitive string
primitive bool
```

(Self-Description)

Schema Grammar

```
start S
S ::= [Schema] types:T*
T ::= P | C
P ::= [Primitive] "primitive" name:sym
C ::= [Class] "class" name:sym ("<" S+)? fields:F*
S ::= <root.classes[it]>
F ::= [Field] name:sym ":" type:<types[it]> M? A?
M ::= "*" { many and optional }
    | "?" { optional }
    | "+" { many }
A ::= "/" inverse:<this.type.fields[it]>
    | "=" computed:Expr
```

Grammar Grammar

```
start G
G   ::= [Grammar] "start"  start:</rules[it]> rules:R*
R   ::= [Rule]          name:sym "::=" arg:A
A   ::= [Alt]           alts:C+ @"|"
C   ::= [Create]      "[" name:sym "]" arg:S  |  S
S   ::= [Sequence] elements:F*
F   ::= [Field]         name:sym ":" arg:P      |  P
P   ::= [Lit]           value:str
      |  [Value] kind:("int" | "str" | "real" | "sym")
      |  [Ref]    "<" path:Path ">"
      |  [Call]   rule:</rules[it]>
      |  [Code]  "{" code:Expr "}"
      |  [Regular] arg:P "*" Sep?     { optional && many }
      |  [Regular] arg:P "?"          { optional }
      |  "(" A ")"
Sep    ::= "@" sep:P
```

non-terminal name
→ reference to rule

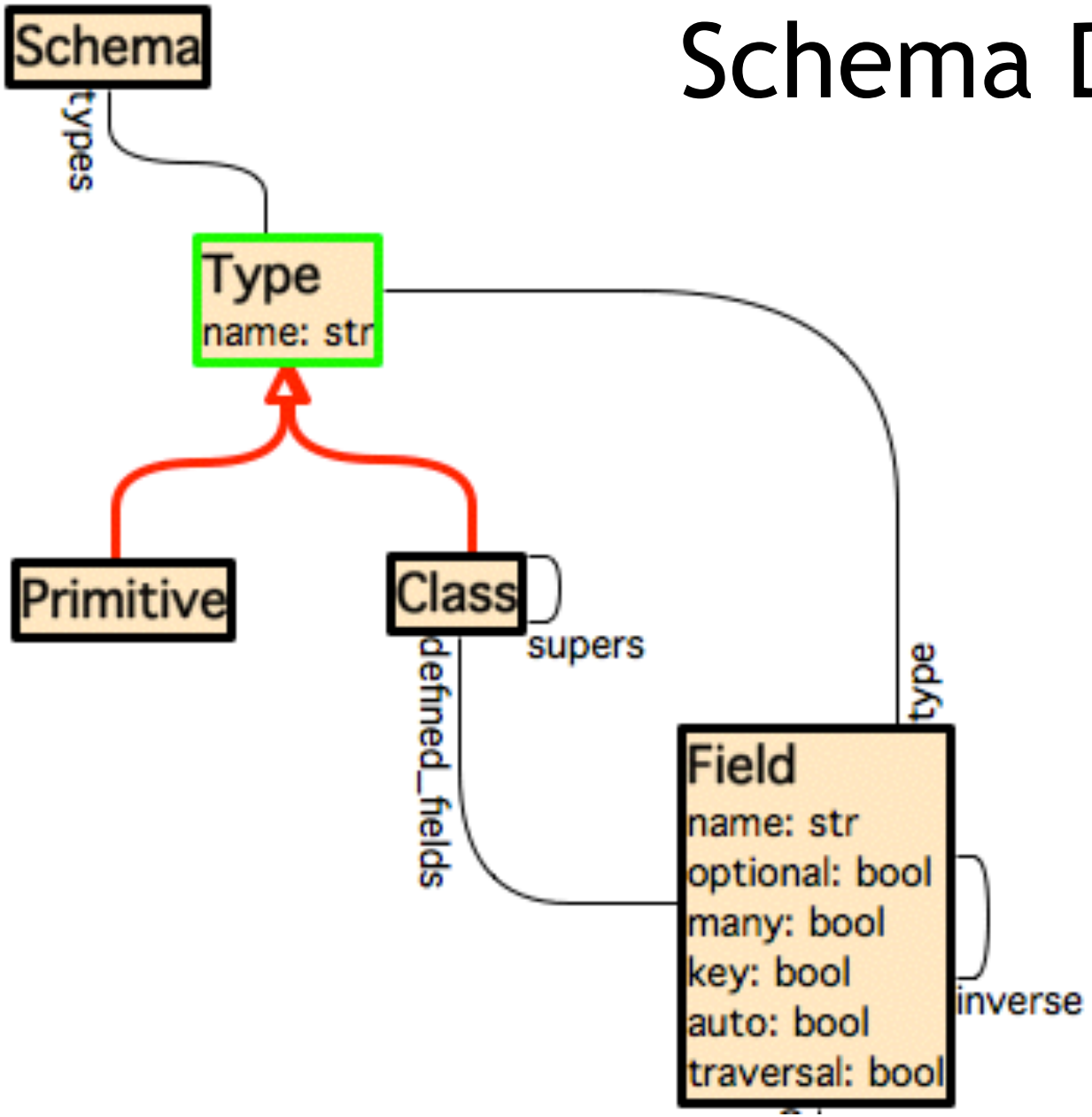
Grammar Schema

```
class Grammar     start: Rule      rules: Rule*
class Rule        name: str        arg: Pattern
class Pattern
class Terminal < Pattern
class Lit < Terminal              value: str
class Value < Terminal            kind: str
class Ref < Terminal              path: Expr
class Alt < Pattern               alts: Pattern+
class Sequence < Pattern          elements: Pattern*
class Call < Pattern              rule: Rule
class Create < Pattern            name: str    arg: Pattern
class Field < Pattern             name: str    arg: Pattern
class Regular < Pattern           arg: Pattern sep: Pattern ?
                                  optional: bool  many: bool
```

Diagrams

- Model
  - Shapes and connectors
- Interpreter
  - Diagram render/edit application
  - Basic constraint solver

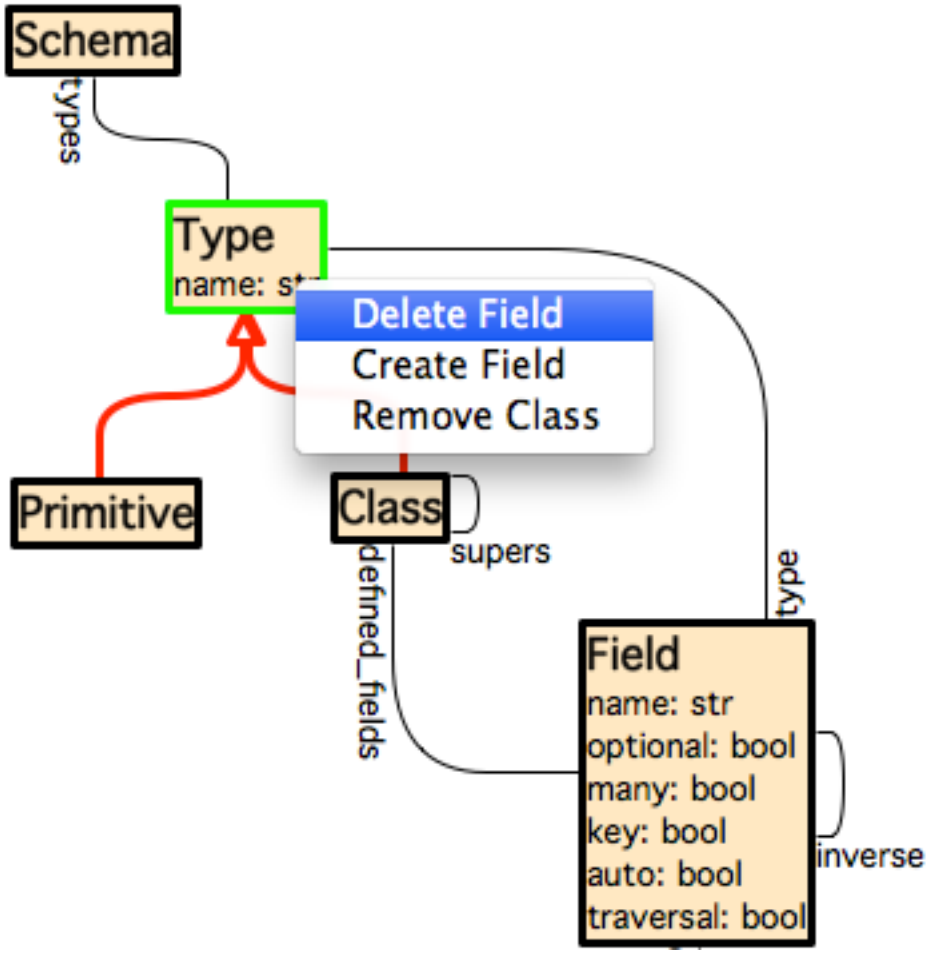
Schema Diagram



Stencils

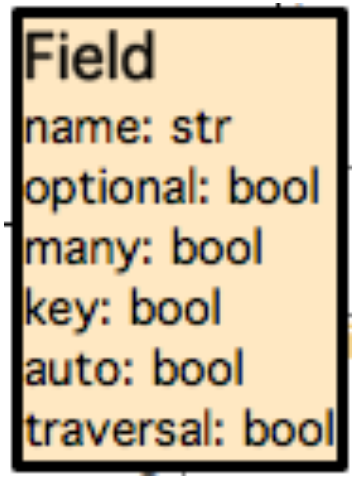
- Model: mapping object graph → diagram
- Interpreter
  - Inherits functionality of Diagram editor
  - Maps object graph to diagram
    - Update projection if objects change
  - Maps diagram *changes* back to object graph
  - Binding for data and collections
    - Strategy uses schema information
    - Relationships get drop-downs, etc
    - Collections get add/remove menus

Schema Diagram Editor



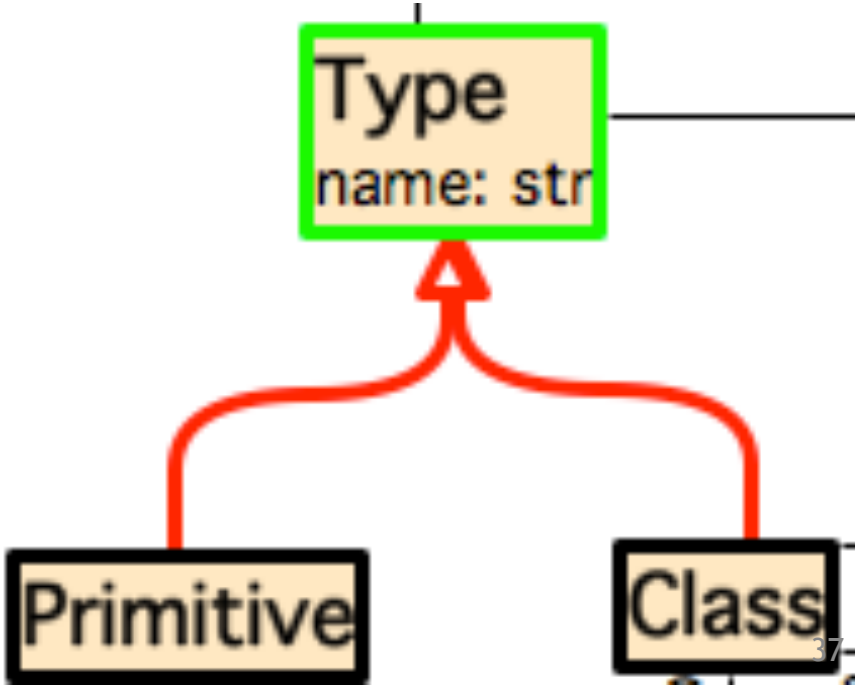
Schema Stencil

```
diagram(schema)
graph [font.size=12,fill.color=(255,255,255)] {
for "Class" class : schema.classes
   label class
     box [line.width=3, fill.color=(255,228,181)] {
       vertical {
          text [font.size=16,font.weight=700] class.name
          for "Field" field : class.defined_fields
            if (field.type is Primitive)
               horizontal {
                 text field.name // editable field name
                 text ": "
                 text field.type.name // drop-down for type
               }}}}
```



Schema Stencil: Connectors

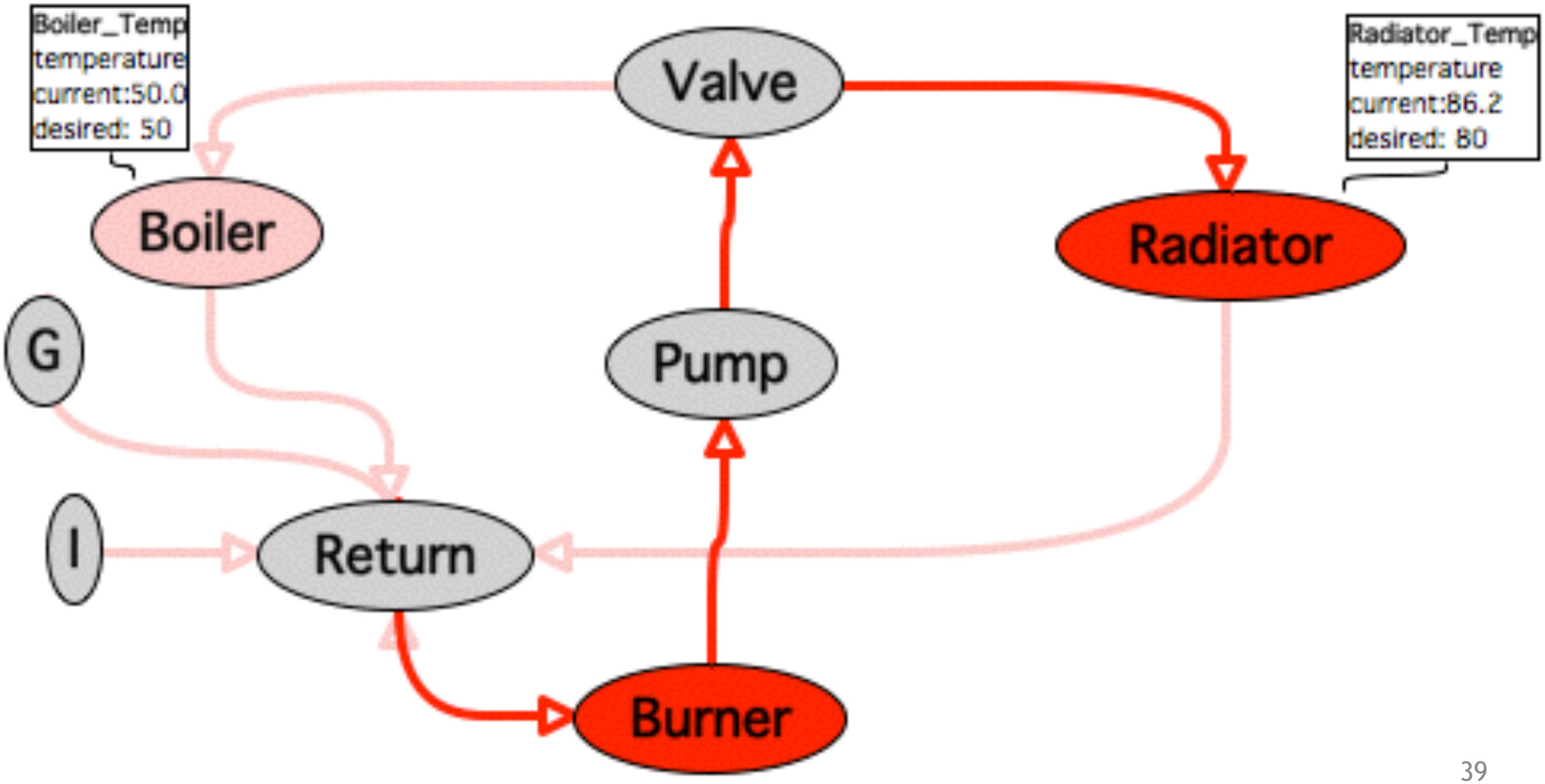
```
 …
// create the subclass links
  for class : schema.classes
    for "Parent" super : class.supers
      connector [line.width=3, line.color=(255,0,0)]
          (class --> super)
…
[also for relationships]
```



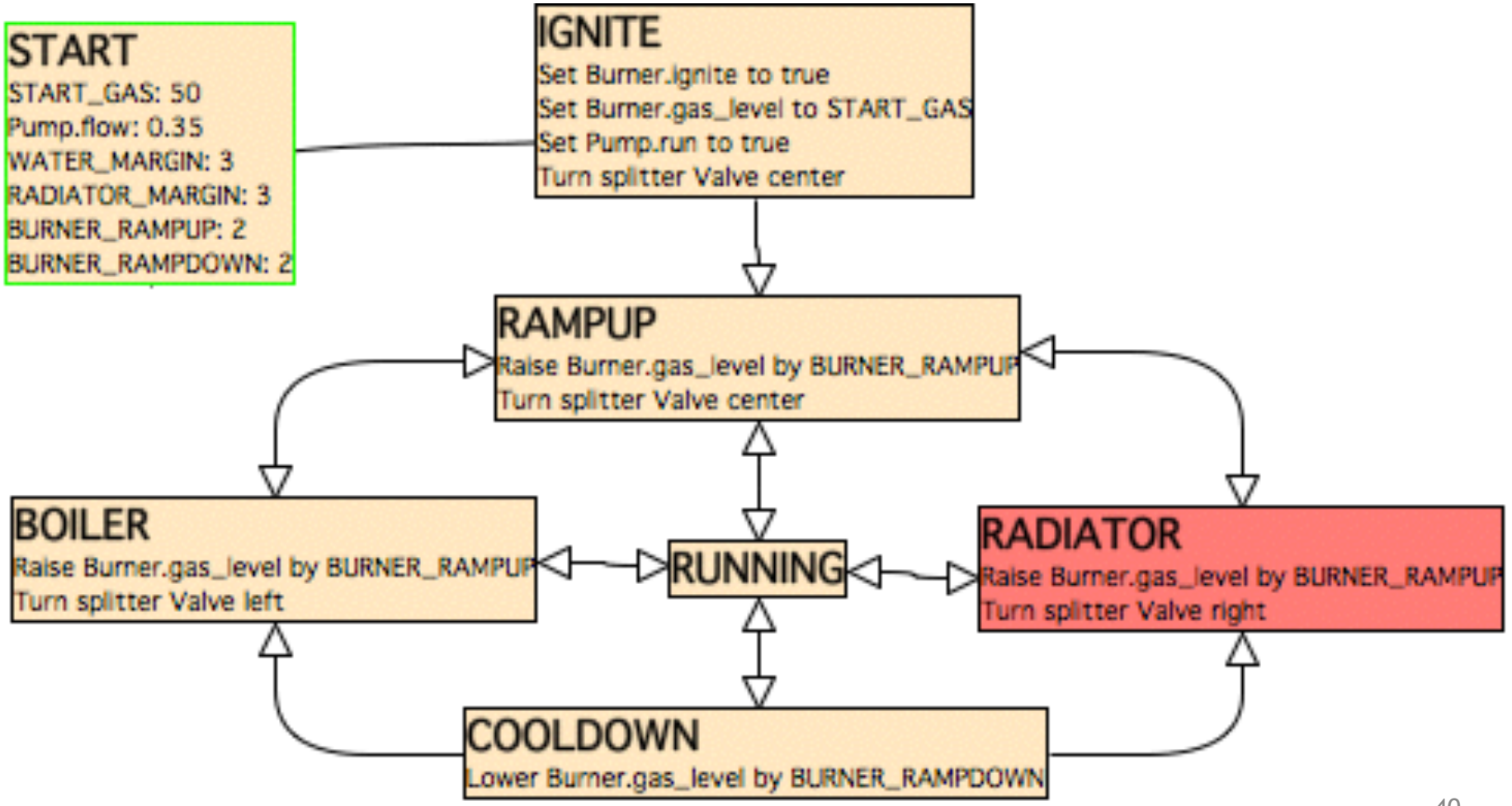
Language Workbench Challenge

- Models
  - Physical heating system
    - furnace, radiator, thermostat, etc
  - Controller for heating system
- Interpreter
  - Simulator for heating system
    - pressure, temperature
  - State machine interpreter
    - Events and actions

Physical Heating System Model



Piping Controller



Piping Details

- Simulation updates physical model
  - Change to physical model causes update to view
  - Observable Data Manager -> Presentation update
- State machine interpreter changes states
  - Presentation shows current state
- User can interact with physical model
  - Change thermostat
- User can edit diagram

Performance

- Ensō is currently slow but usable
  - Accessing a field involves two levels of meta-interpretation
  - My job is to give compiler people something to do
- Partial Evaluation of model interpreters

```
web(UI, Schema, db, request) : HTML
web[UI, Schema](db, request) : HTML
```

static

dynamic

Aspect	Code SLOC	Model SLOC
Bootstrap	387	—
Utilities	256	—
Schemas	691	51
Grammar/Parse	885	106
Render	318	17
Web	932	305
Security	276	46
Diagram/Stencil	1389	176
Expressions	448	144
Core	5582	844
Piping	527	268

Ensō Summary

- Executable Specification Languages
  - Data, grammar, GUI, Web, Security, Queries, etc.
- External DSLs (not embeded)
- Interpreters (not compilers/model transform)
  - Multiple interpreters for each languages
- Composition of Languages/Interpreters
  - Reuse, extension, derivation (inheritance)
- Self-implemented (Ruby for base/interpreters)
  - Partial evaluation for speed

Related Work

- Aspects: a fundamental idea
  - Current solutions are terrible (AspectJ)
- DSLs and Models: Feeling same elephant
  - external vs. internal
  - graphical vs. textual

Language	Meta-Model
Program	***Model***

- F# Type Providers
- Scheme macros (defstruct)
- Metaprogramming
  - But without manipulating 'code'

Don't Design Your Programs

Program
Your Designs

Ensō
enso-lang.org