Keynote, SLE’25

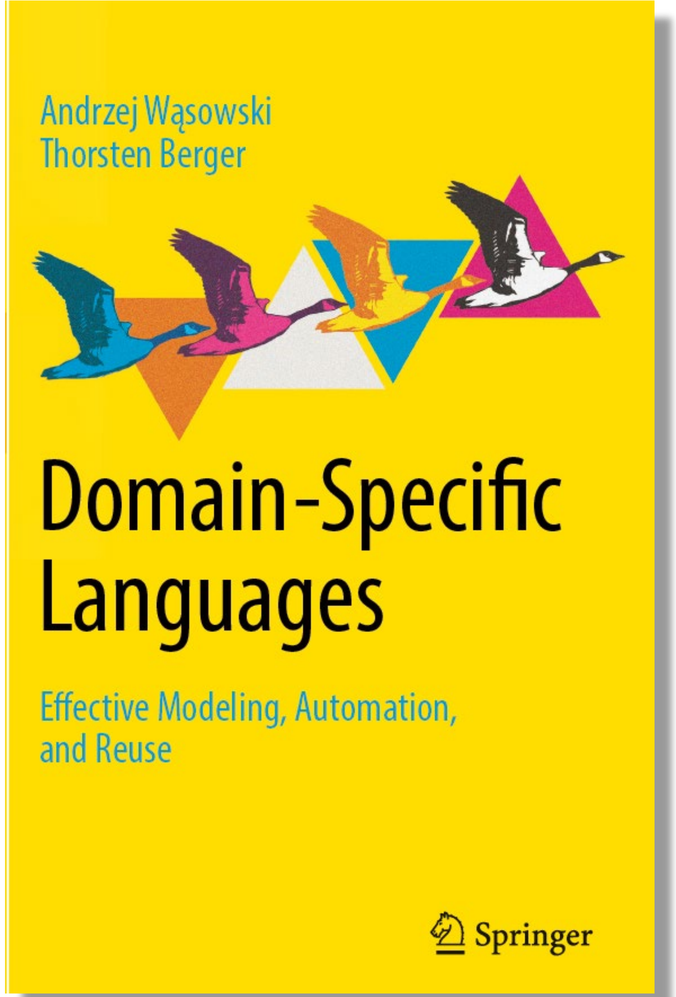
a new DSL textbook in town!

thorsten berger

RUHR UNIVERSITÄT BOCHUM
RUB

IT University of Copenhagen

Andrzej Wasowski



http://dsl.design

RUHR UNIVERSITÄT BOCHUM RUB

Thorsten Berger

Skip Ad ▶|

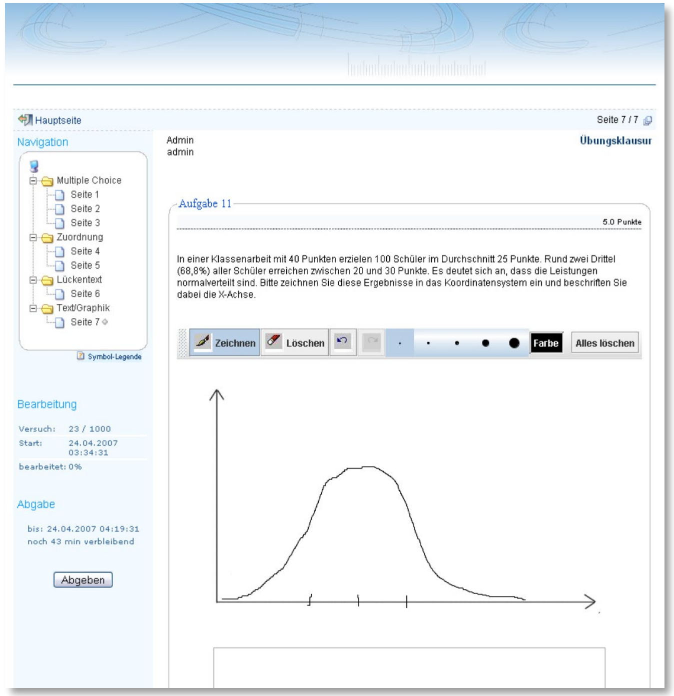
my interest in DSLs

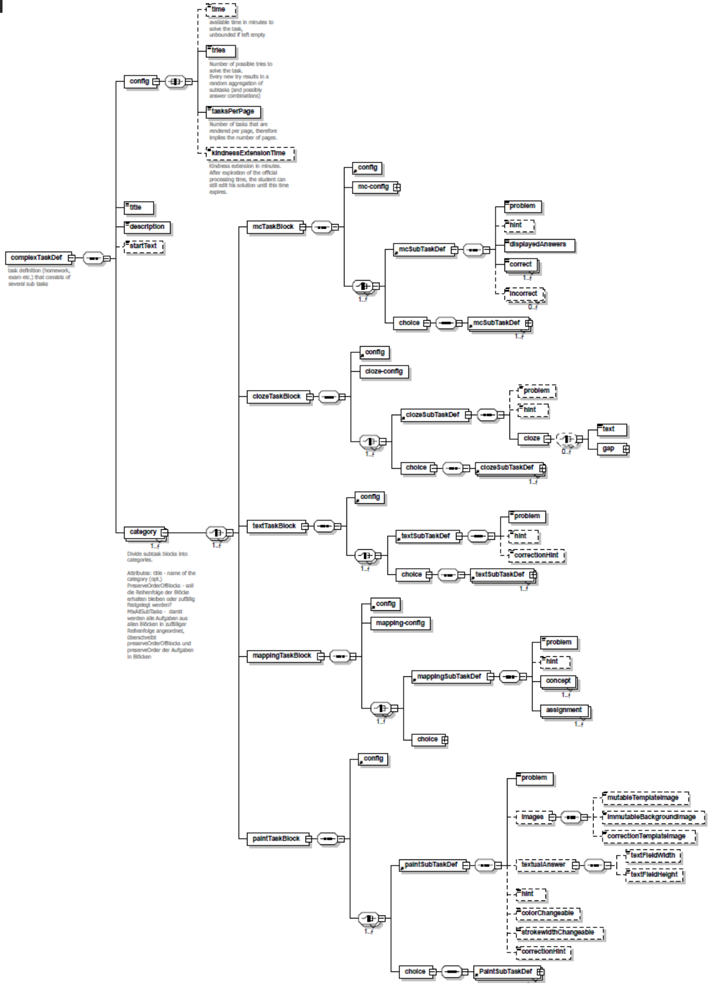
2007 student job: developed eAssessment software

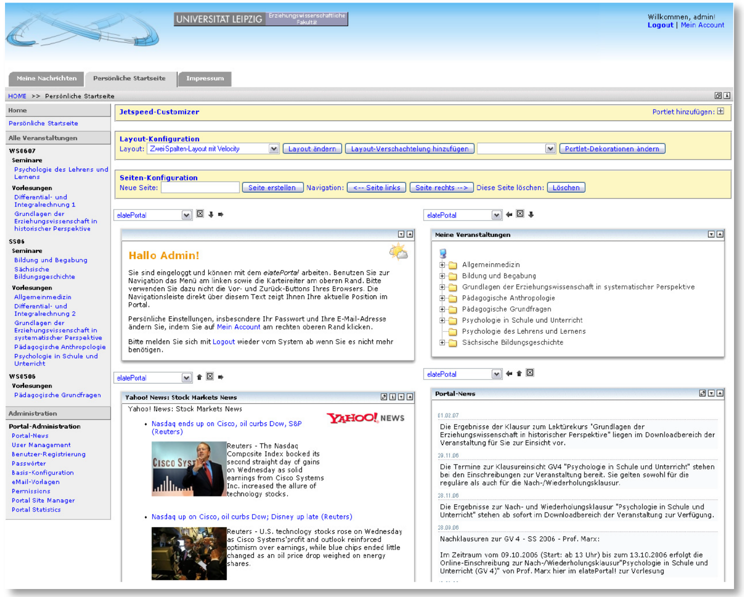
- customizable web app (struts), portal server (jetspeed-2), authoring tools

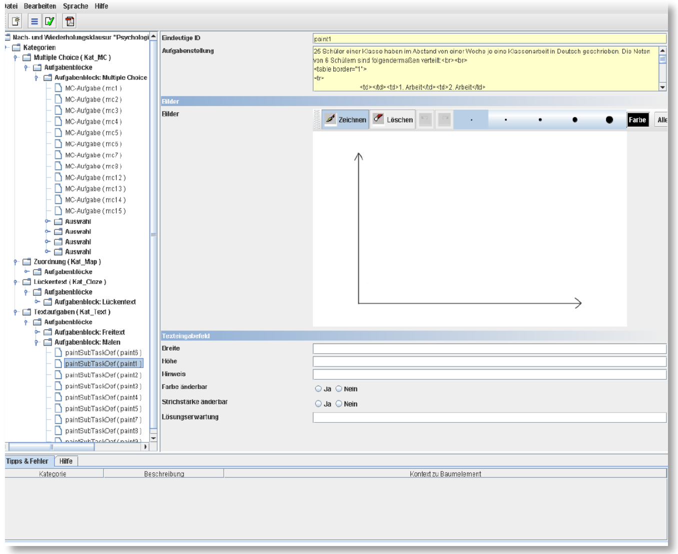
using model-driven technology

- Eclipse EMF with GEF, and JAXB





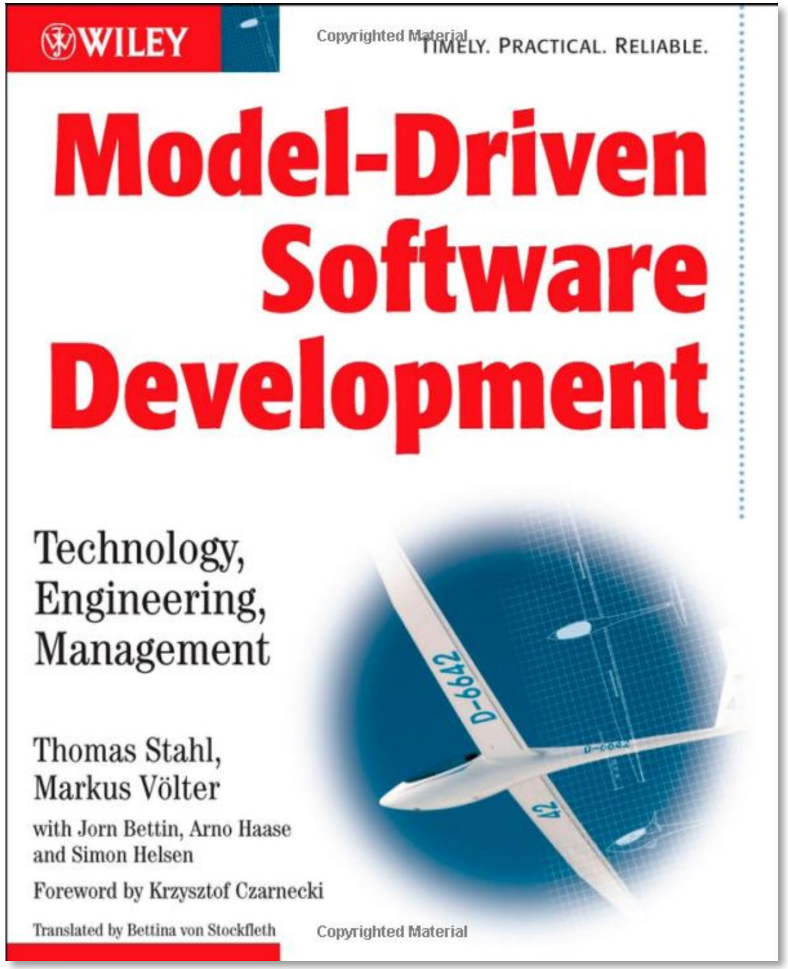




back at that time

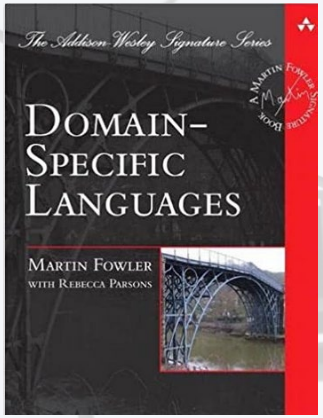
[Stahl and Völter 2006]

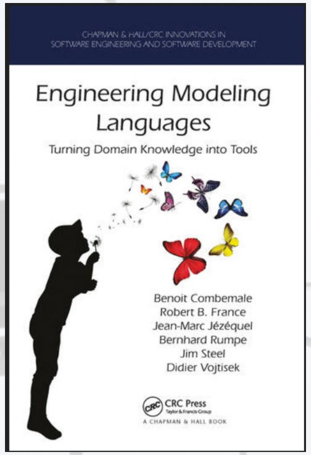
- likely the most-referenced book on MDSE
- helped to establish MDSE as a field
- made it known to practitioners
- UML-based approach to DSLs

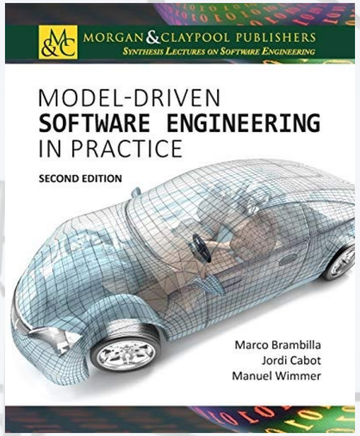


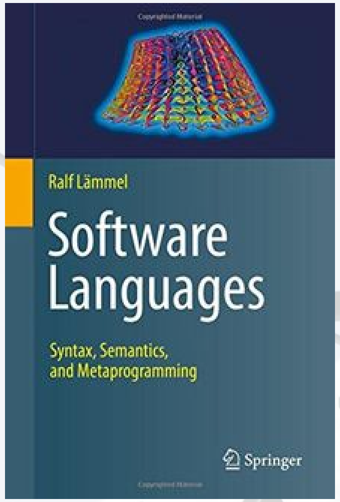
more DSL books

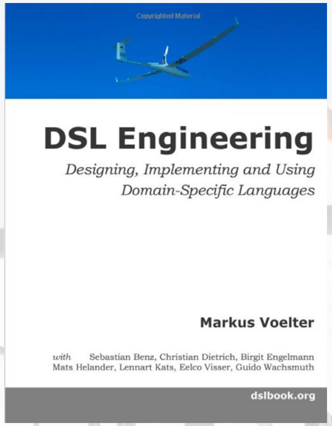
thanks for the foreword, Ralf!

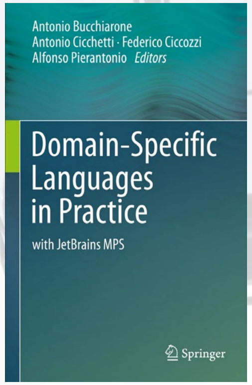


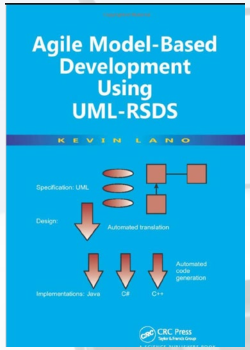


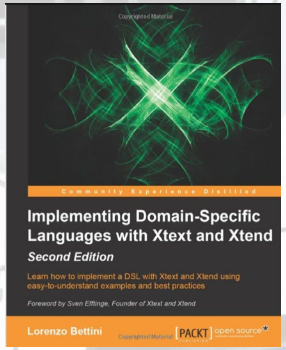


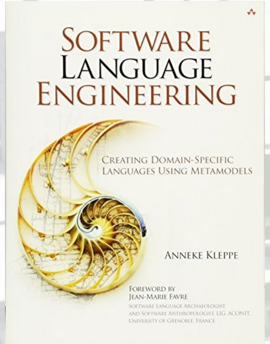


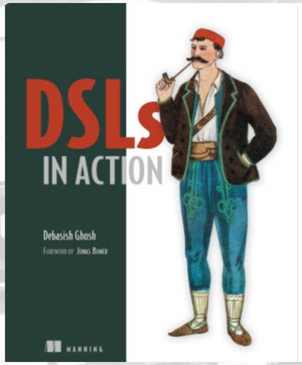


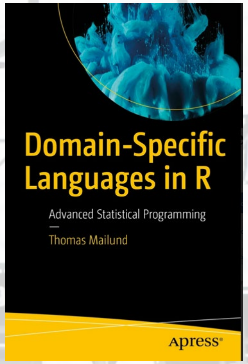












standards and frameworks

Object Management Group (OMG)

Eclipse Foundation

allowed building large-scale safety-critical software

www.vsd-project.org

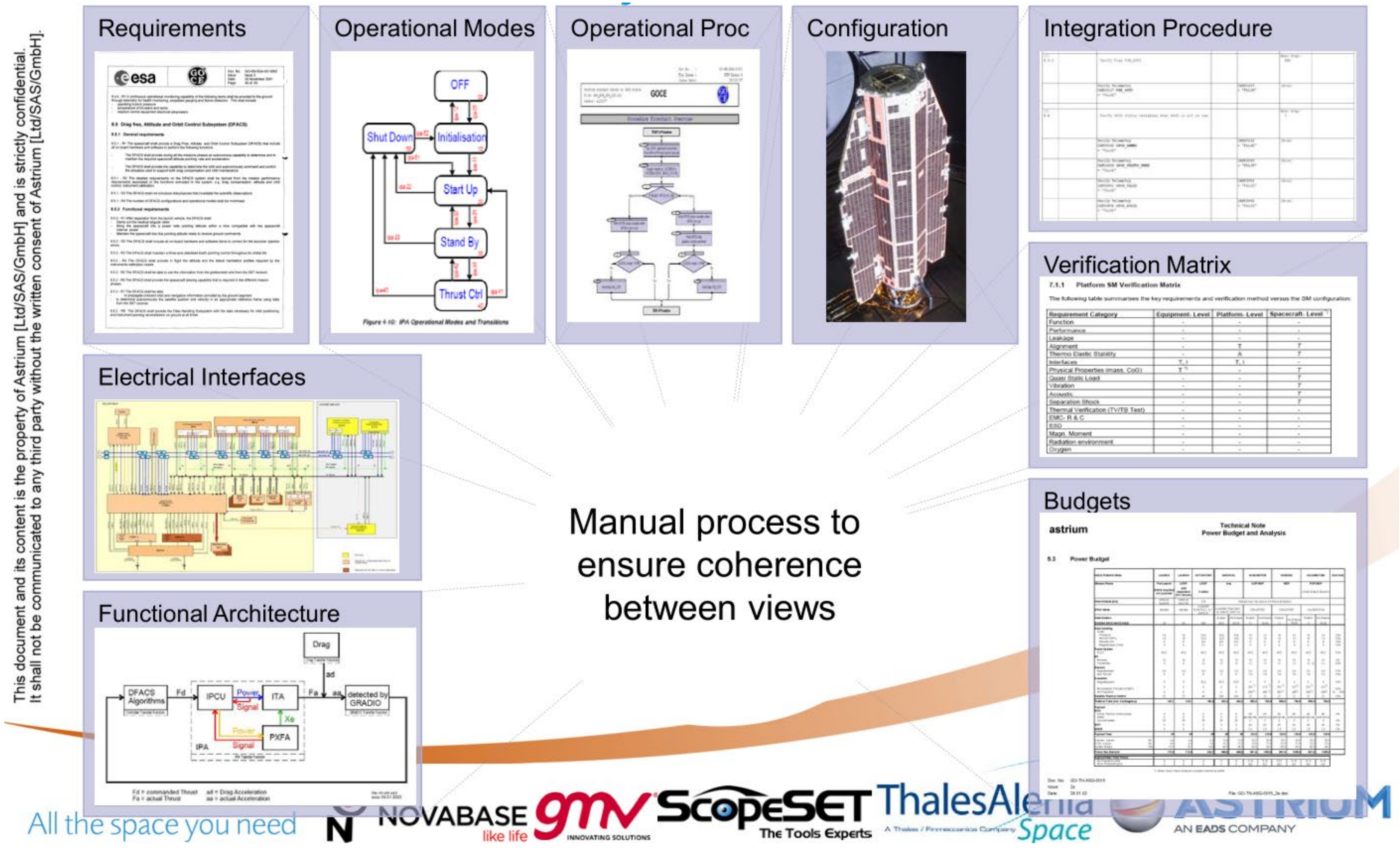
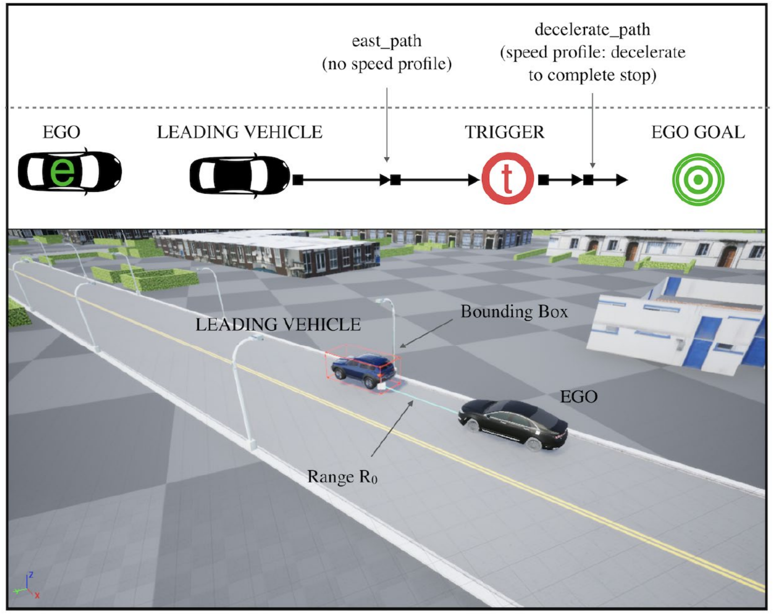


figure from: https://www.vsd-project.org/download/presentations/VSD_P2_FP_2012-05-15_v3.pdf

DSLs for autonomous driving



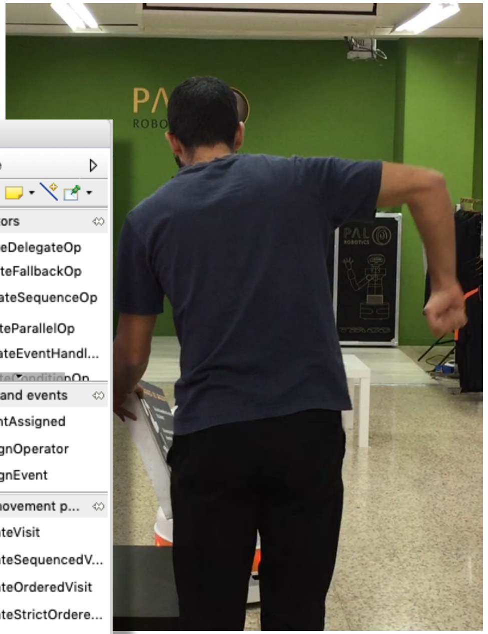
Queiroz, Berger, Czarnecki, "Geoscenario: An Open DSL for Autonomous Driving Scenario Representation," in *30th IEEE Intelligent Vehicles Symposium (IV)*, 2019.

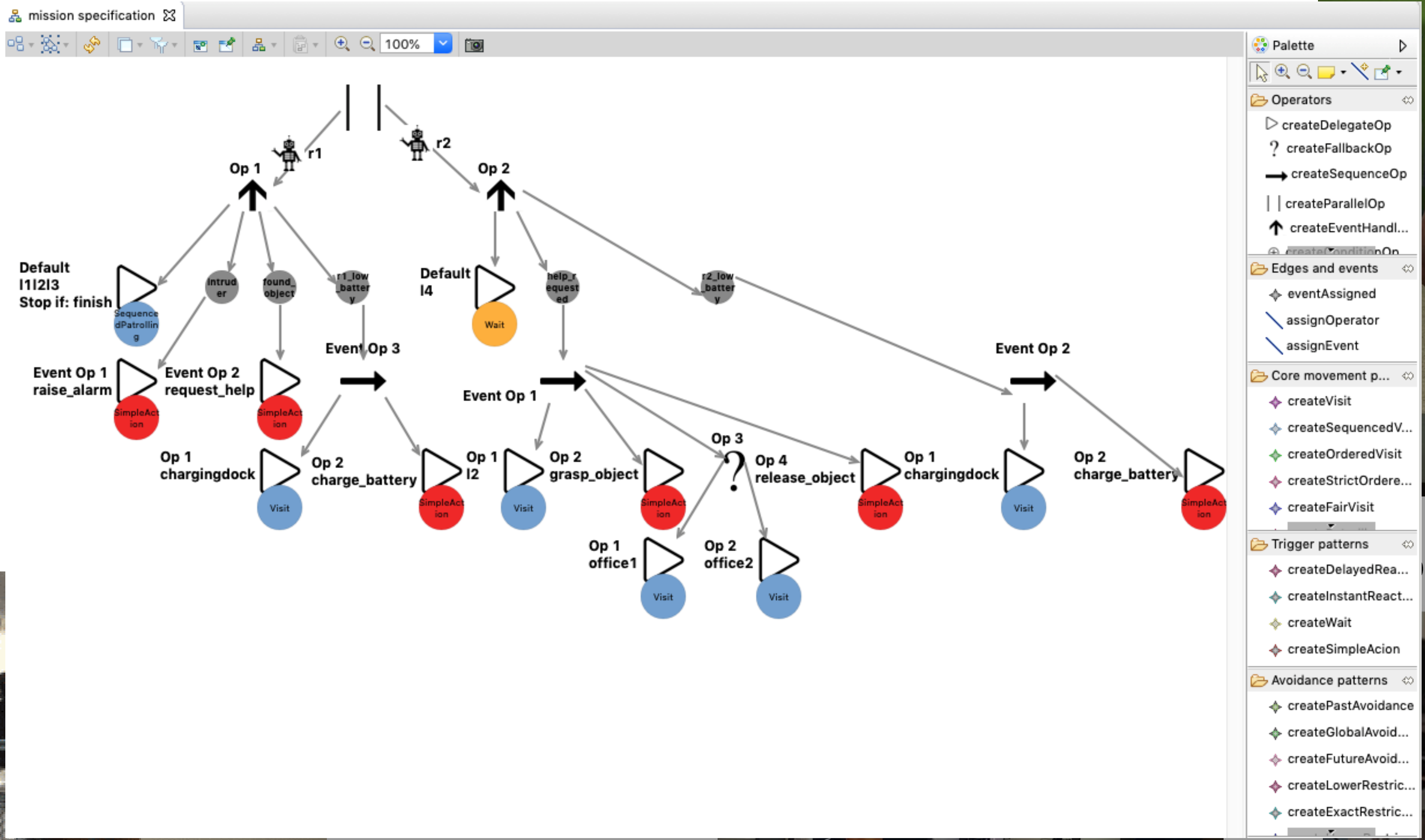
Queiroz, Sharma, Caldas, Czarnecki, Garcia, Berger, Pelliccione, "A Driver-Vehicle Model for ADS Scenario-based Testing," *IEEE Transactions on Intelligent Transportation Systems (T-ITS)*, 2024

4Robots

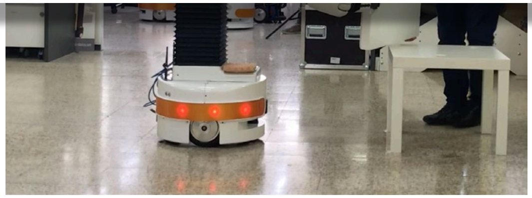
DSLs for robots

gesture recognition





boCup tasks



many different technological spaces

modelware (SE perspective)

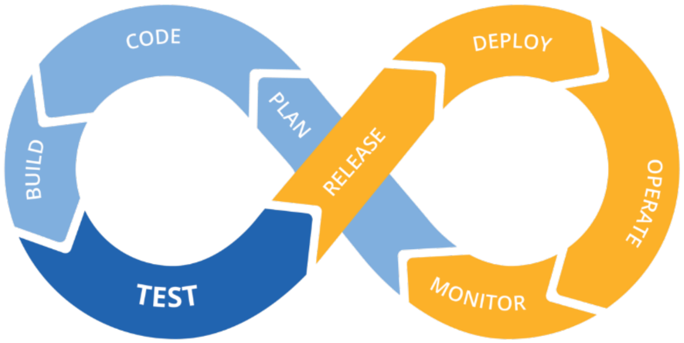
grammarware (PL perspective)

others, according to [Lämmel 2018]

- XMLware (e.g., XML, XML infoset, DOM, DTD, XML Schema, XPath, XQuery, XSLT)
- JSONware (e.g., JSON, JSON Schema, JSONata)
- SQLware (e.g., table, SQL, relational model, relational algebra, WOL),
- RDFware (e.g., resource, triple, Linked Data, RDF, RDFS, OWL, SPARQL, STTL)
- Objectware (e.g., objects, object graphs, object models, state, behavior, visitor pattern)
- Javaware (e.g., Java, Java bytecode, JVM, Eclipse, JUnit)

DSLs in the age of **agile** and **AI**?

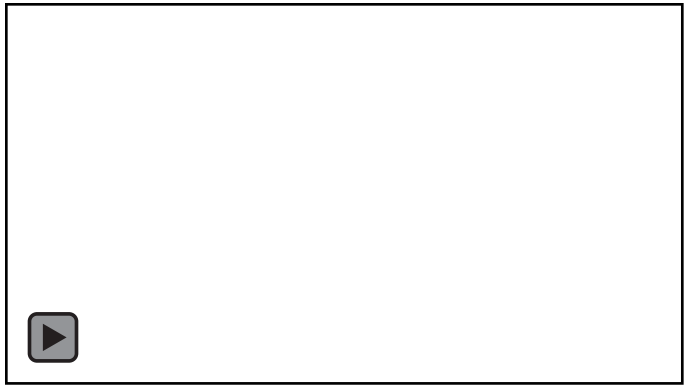
DSLs for automation more important than ever



automation $\xrightarrow[\text{requires}]{}$ **abstraction** $\xrightarrow[\text{requires}]{}$ **language**

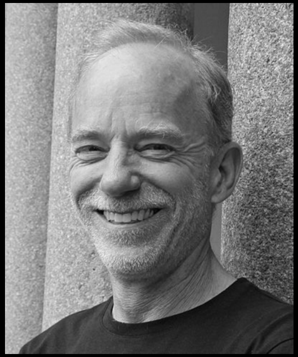
DSLs for assuring guarantees

SE4AI / AI4SE



guarantees $\xrightarrow[\text{require}]{}$ **abstraction** $\xrightarrow[\text{require}]{}$ **language**


Francis-Noël Thomas


Mark Turner

“Language is sufficient to any thought. Imperfect expression is the fault of limited writers, not limited language.”

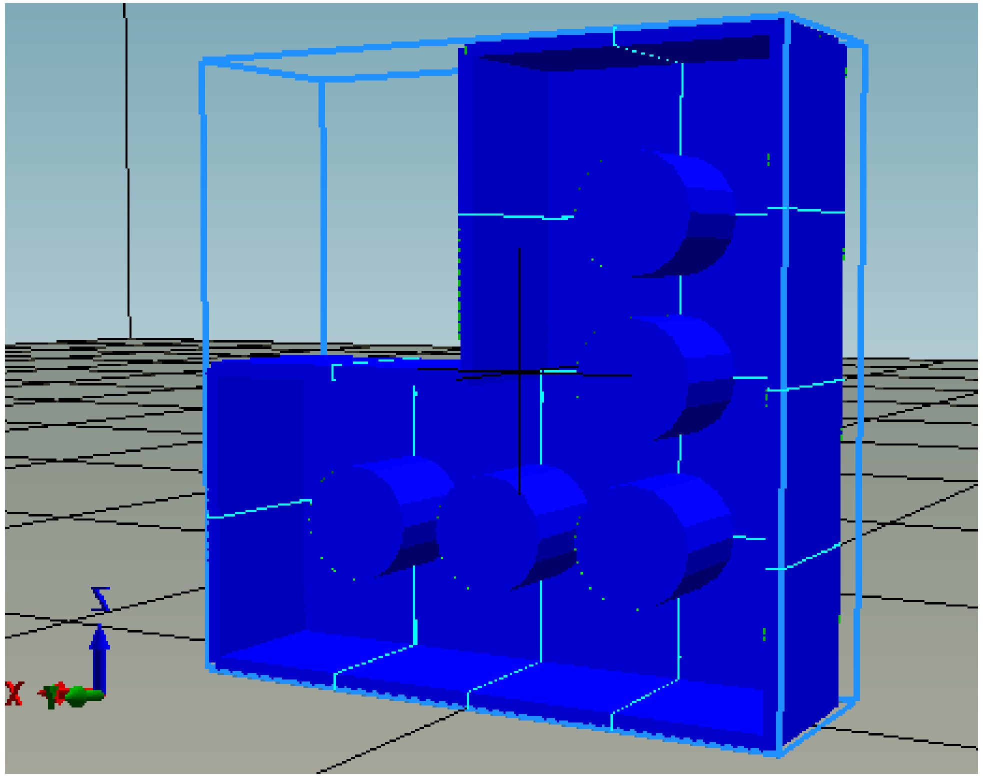
“Parser development is still a black art.”

Paul Klint, Ralf Lämmel, and Chris Verhoef. “Toward an engineering discipline for Grammarware”. ACM Trans. Softw. Eng. Methodol. 2005

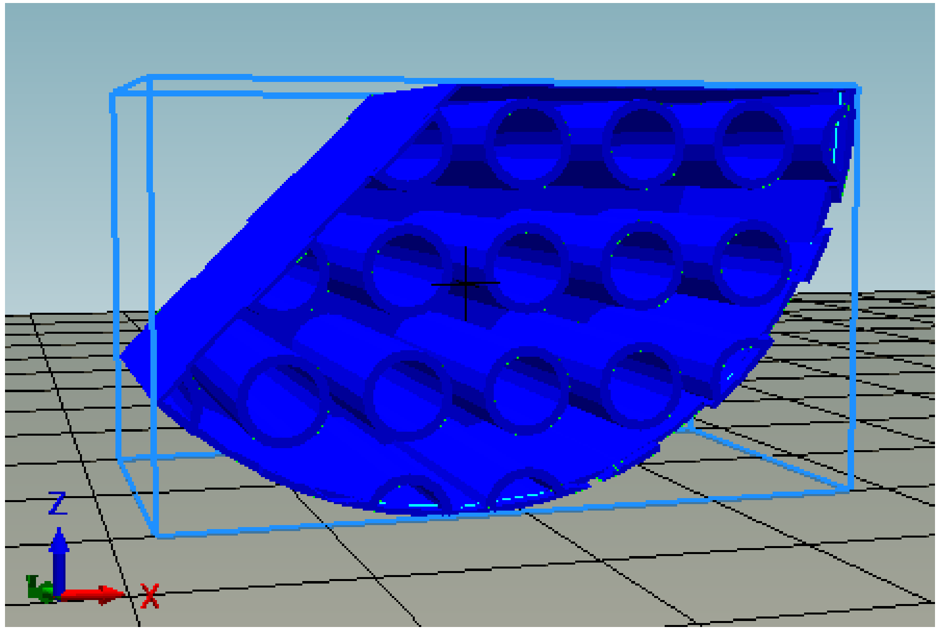
smaller DSLs built in my course

a DSL for Lego bricks

```
{
        "Lego": "Star",
        "Length": 20,
        "Width": 20,
        "Bricks" : [{
                "Brick": "Wars",
                "Width": [4],
                "Length": [2]
        }, {
                "Brick": "Trek",
                "Width": [2],
                "Length": [2]
        }]
}
```

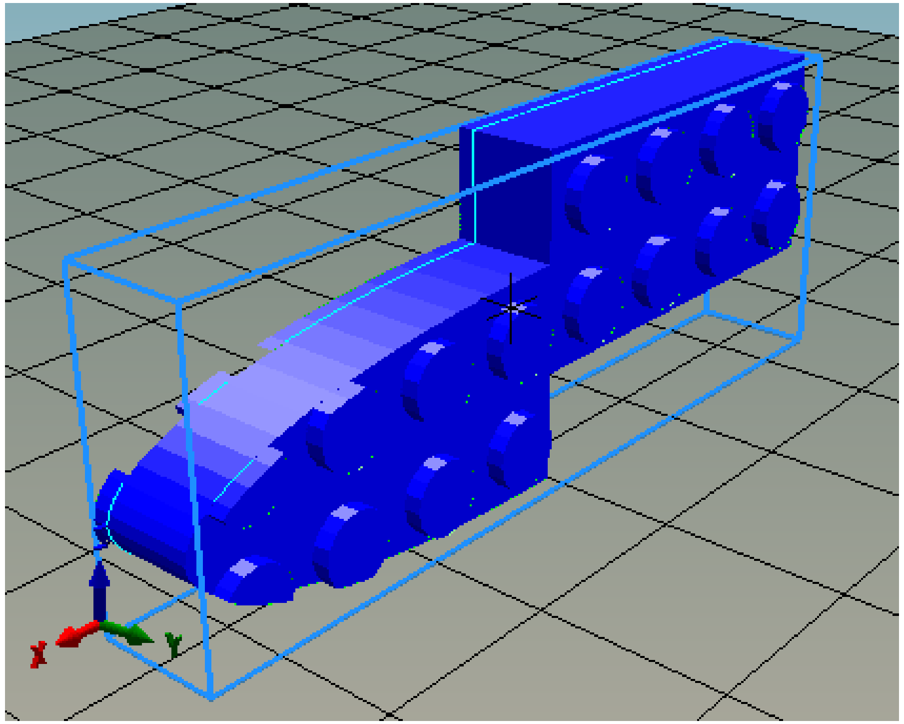


built with different styles of concrete syntax



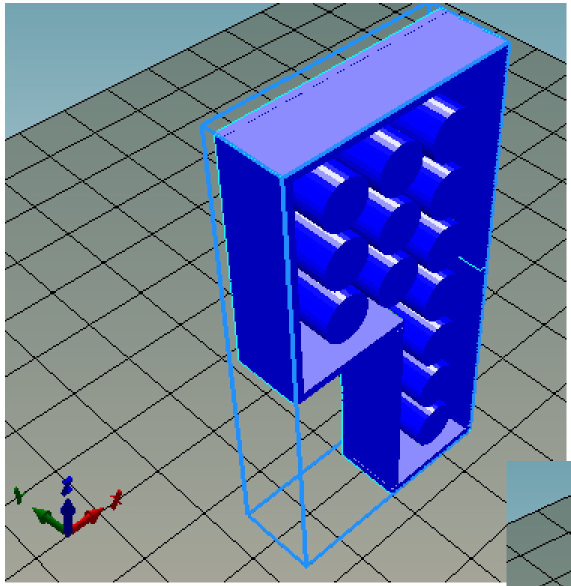

```
AdjLegoSystem {
	thickness 20
	finalBrick Pizza
	abstractlegobrick {
		RoundedBrick Pizza{
			roundedSide ALL
			sizeproperties {
				int length = 7,
				int width = 7
			}
		},
		SlicedBrick Slice {
			portions 3
			brick Pizza
		}
	}

}
```

containing composition operators

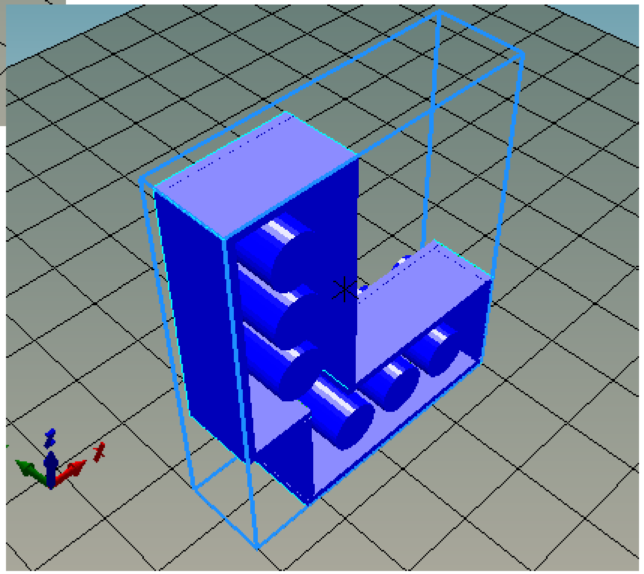


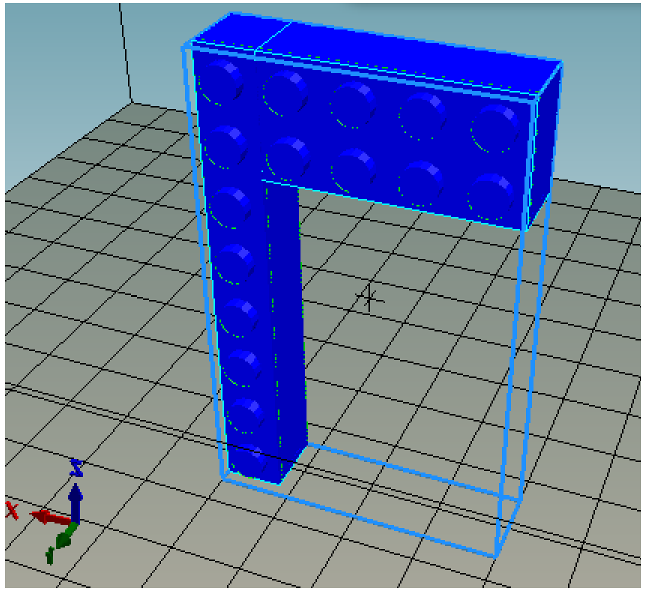

```
AdjLegoSystem {
        thickness 7
        finalBrick Boomerang
        abstractlegobrick {
                RoundedBrick Frisbee{
                        roundedSide RIGHT
                        sizeproperties {
                                int length = 4,
                                int width = 2
                        }
                },

                SquareBrick Stick {
                        sizeproperties {
                                int length = 4,
                                int width = 2
                        }
                },

                Combination Boomerang {
                        mainSide LEFT
                        position 3
                        main Frisbee
                        secondary Stick
                }
}
```

many more styles of concrete syntax



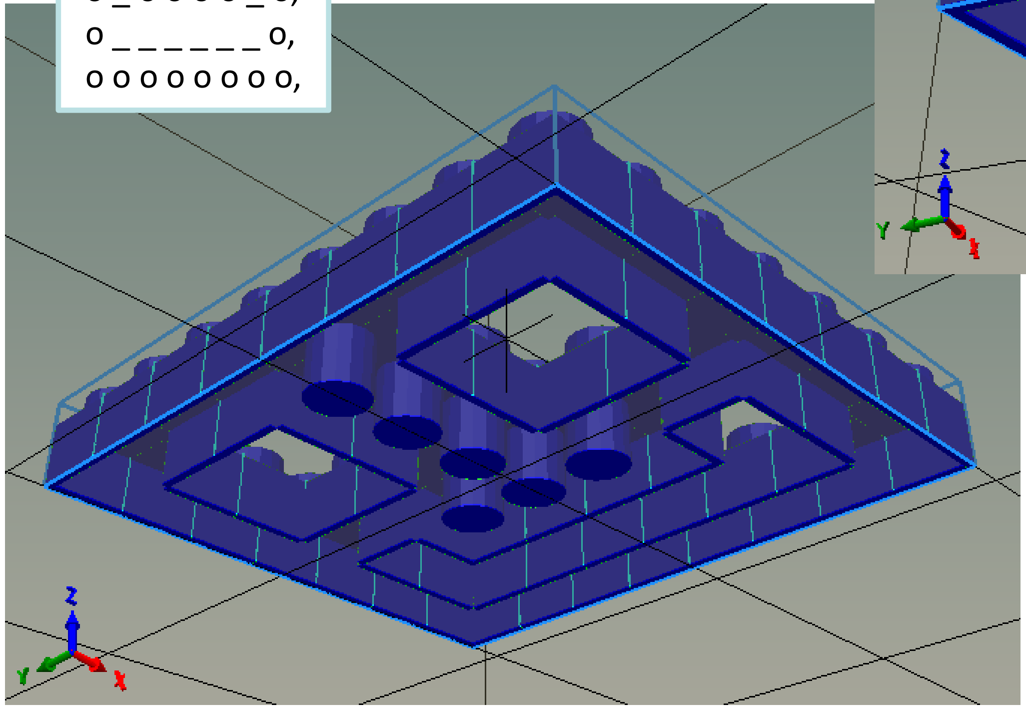
```
dimensions 10 x 10;
"2x4": 2 x 4;
"4x2": 4 x 2;
"1x8": 1 x (2 * "4x4".width);
"4x4": "2x4".height x (2 * "2x4".width);
"Composite Brick 1": "2x4" <- "4x4" TOP: LEFT 1 <- LEFT 1;
"Composite Brick 2": "2x4" <- "4x2" BOTTOM: LEFT 1 <- LEFT 4;
"Composite Brick 3": "1x8" <- "4x2" RIGHT: TOP 1 <- BOTTOM 2;
```

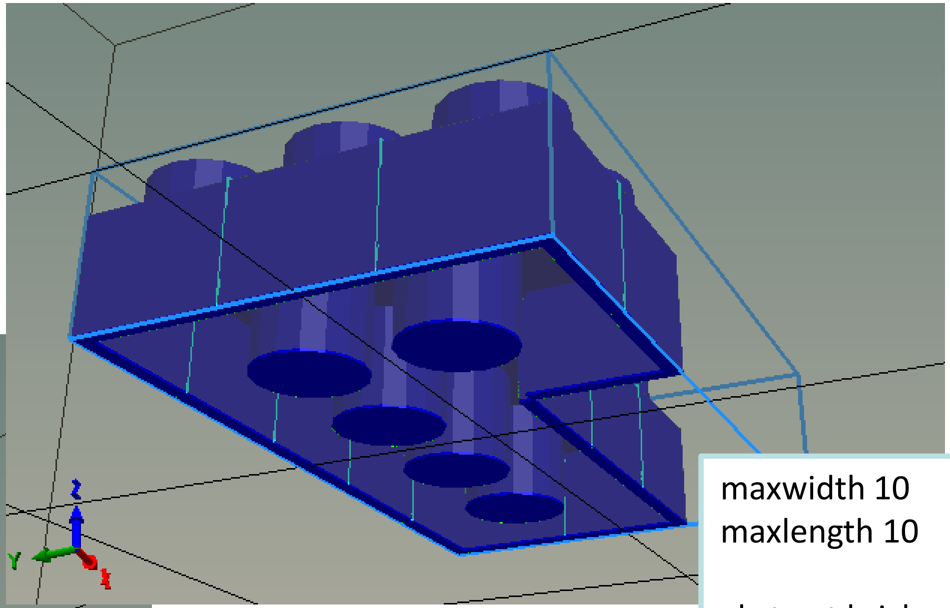




unleashing creativity

```
brick smiley
o o o o o o o o o,
o _ _ o o _ _ o,
o _ _ o o _ _ o,
o o o o o o o o o,
o _ o o o o _ o,
o _ _ _ _ _ _ o,
o o o o o o o o o,
```





```
maxwidth 10
maxlength 10

abstract brick a
  o o o,
  o o o,

abstract brick b
  _ o o,
  _ o o,
  _ o o,

combo T a over b
```

So how do we teach DSL engineering?

How can our book help?

highlights

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

concrete exercises: train building low-level skills

PL perspective linked and mixed with SE perspective

teaching to test is teaching to build

teach small and larger DSLs

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

concrete exercises: train building low-level skills

PL perspective linked and mixed with SE perspective

teaching to test is teaching to build

teach small and larger DSLs

finite state machines as a DSL

domain analysis to identify concepts and relationships

formalize in a **meta-model**

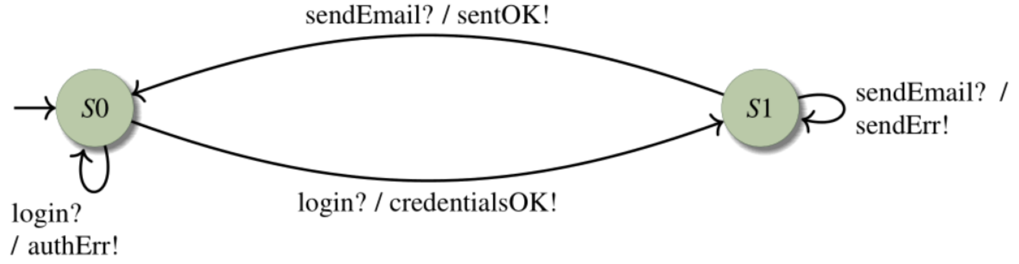
Q1: Purpose To build examples of student exercises; To interact with examples using an interpreter, an interpreter will be needed.

Q2: Users Computer science students learning automata theory (probably knowing the basics of a programming language); A professor, who can provide the examples and will ask the students to use the tool.

Q3: Concepts Finite-state machines, several in parallel; States; Transitions;

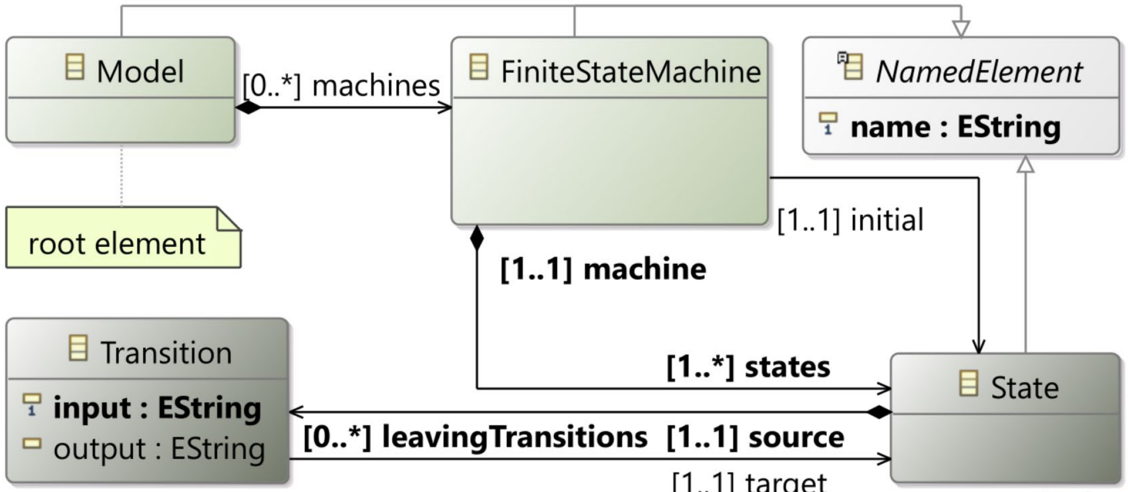
Q4: Relations *Properties:* states may be initial or end states, states and machines have names, transitions have input action labels, transitions have optional output labels; *Relations:* machines *own* states, transitions *connect* source and target states.

Q5: Examples The professor whose students are supposed to use the tool provided us with the following example of a model in concrete graphical syntax:



finite state machine (FSM) DSL

abstract syntax



source: fsm/model/fsm.ecore

principles

guideline 3.1: create a single partonomy

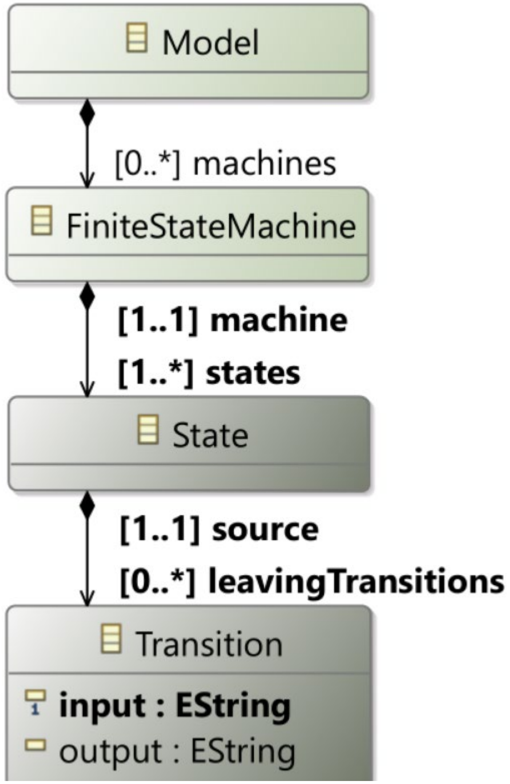
guideline 3.2: avoid interfaces and methods

...

guideline 3.7: let the meta-model describe the problem, not the software tool solving it

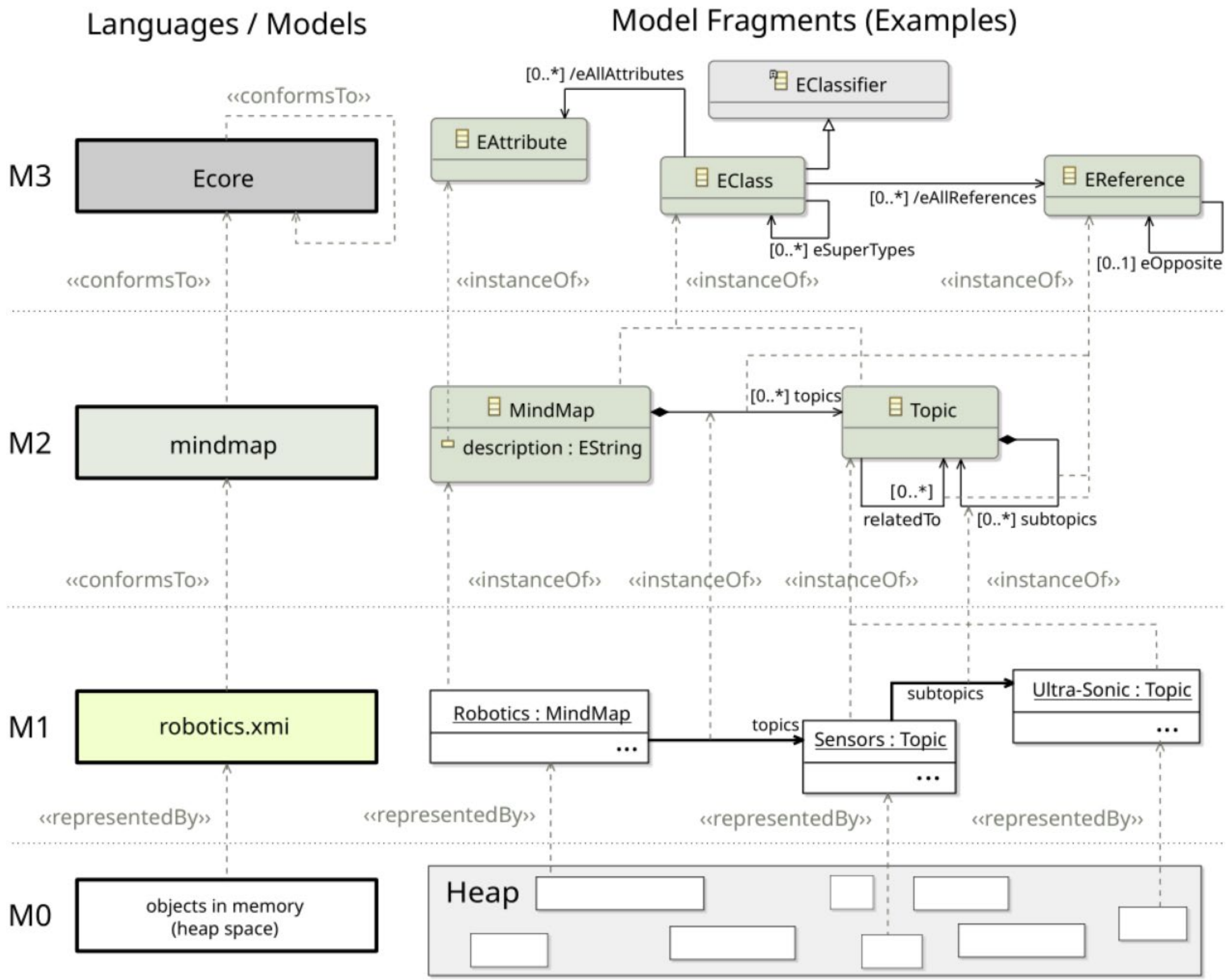
guideline 3.8: avoid scope creep

...



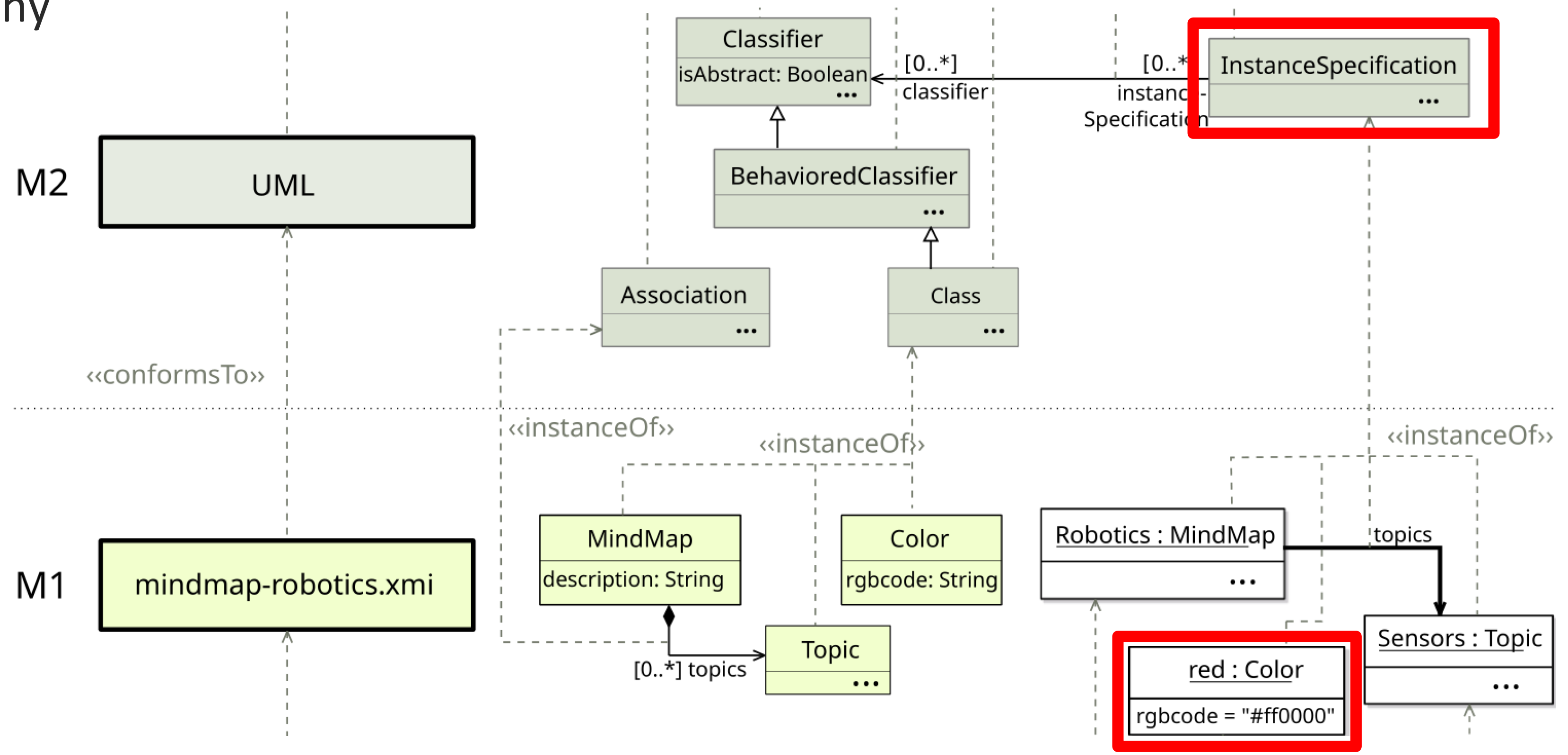
partonomy of FSM

meta-modeling hierarchy



multi-level modeling?

a concept that is now easy to explain upon the meta-modeling hierarchy



concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

easier to remember, easier to understand

concrete exercises: train building low-level skills

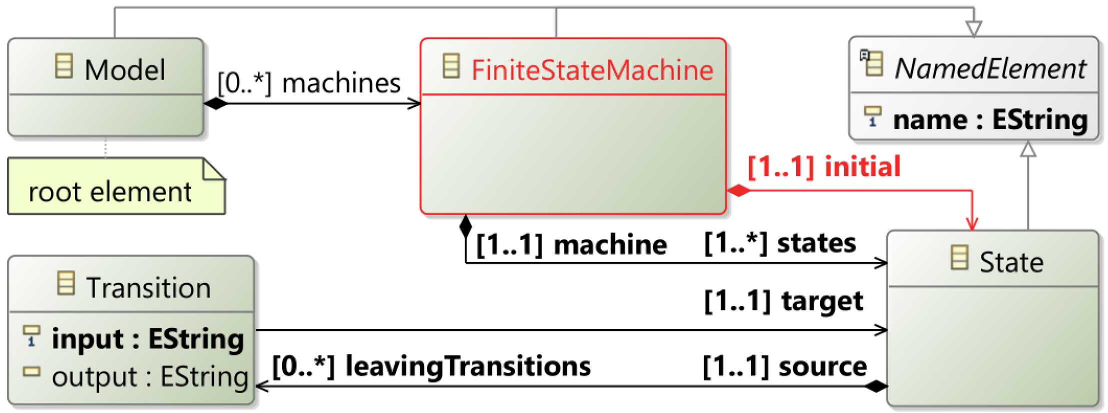
PL perspective linked and mixed with SE perspective

teaching to test is teaching to build

teach small and larger DSLs

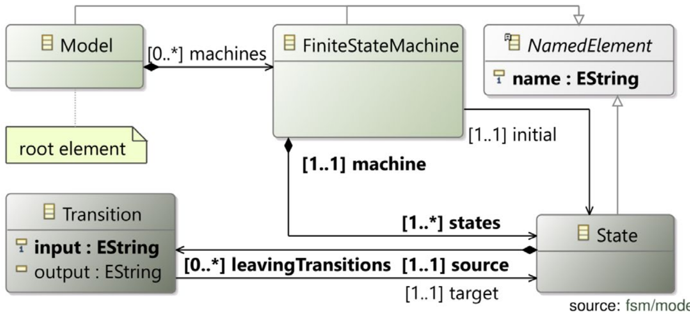
quality assurance of meta-models

Definition 3.4. *A meta-model is **consistent** if it can be instantiated meeting all constraints of the meta-modeling language semantics. A meta-model is **element-consistent** if for each element of the meta-model there exists an instance in which this element is instantiated.*



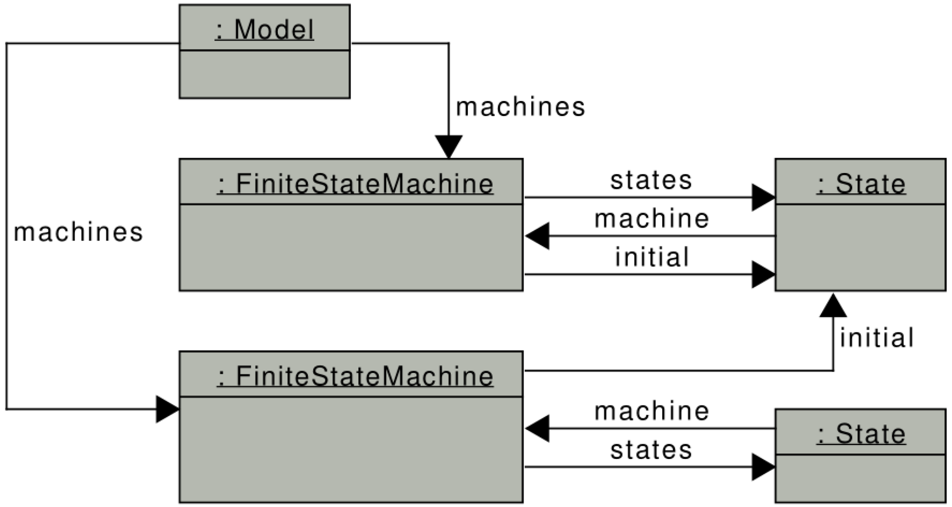
*Definition 3.5. A meta-model is **parsimonious** if it contains no meta-classes, no relations (references, associations), and no attributes that do not address any system requirements for the modeling language.*

static semantics of DSLs



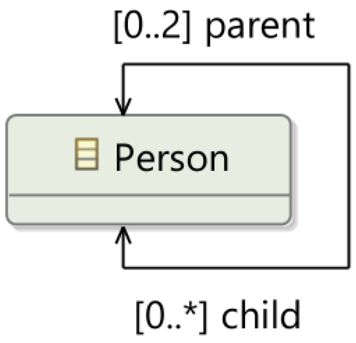
source: fsm/model/fsm.ecore

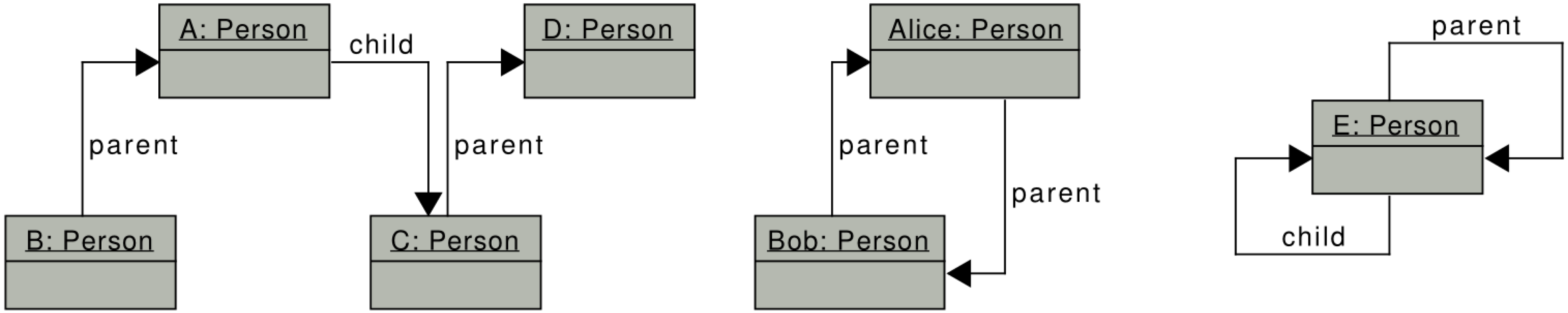
an unexpected instance



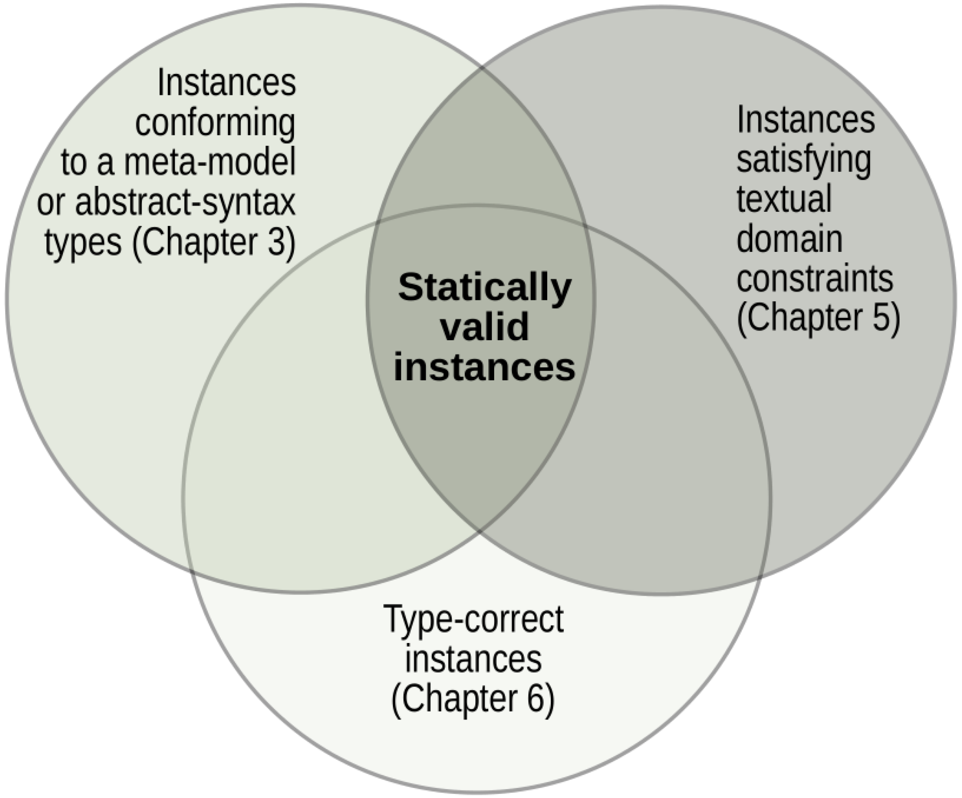
Definition 5.3. *A* ***constraint*** *is a pure (side-effect free) Boolean expression declared over elements of a meta-model, but interpreted over its instances. Its purpose is to restrict the set of valid instances of the meta-model.*

more illustrative examples





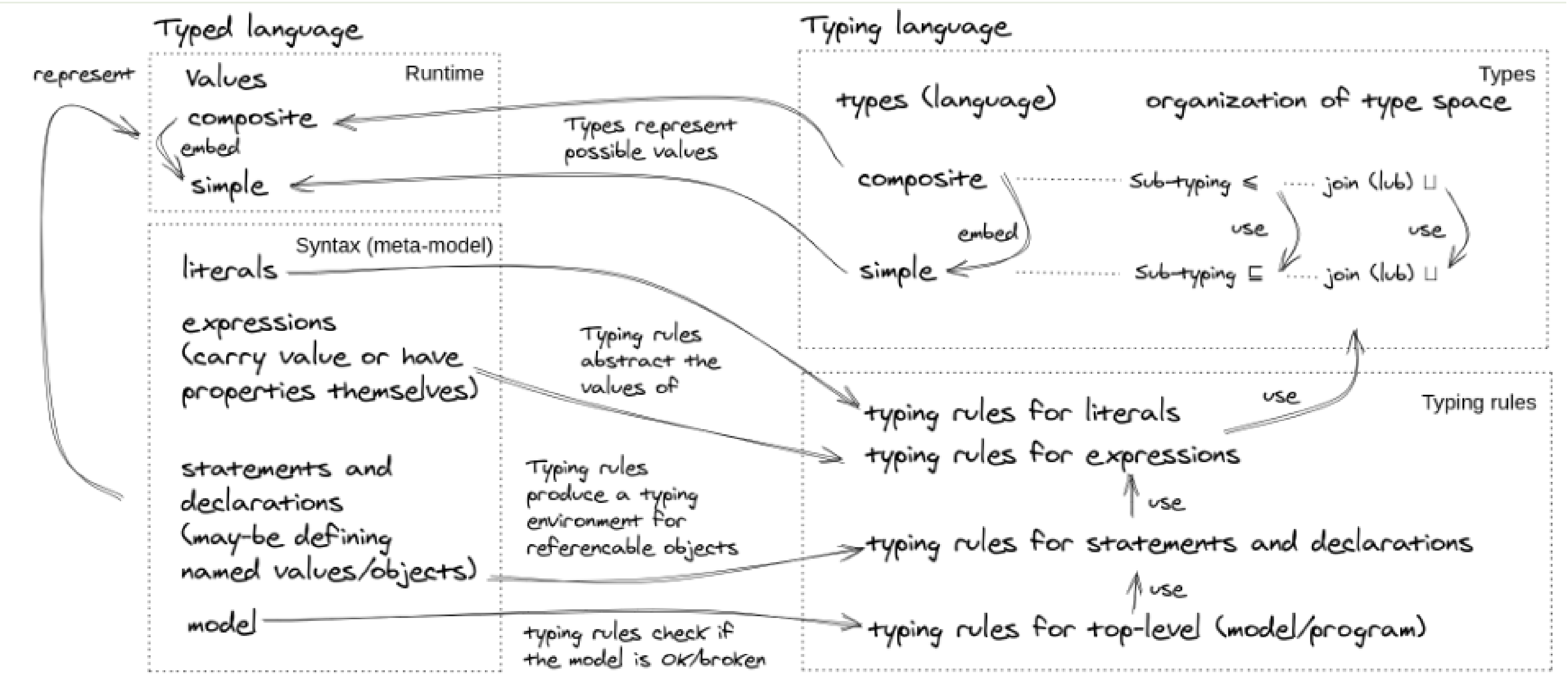
static semantics: meta-model + domain / type constraints



ingredients of a type system

unlike a typical constraint, a type system examines the entire instance, not just few related objects

- may overwhelm when compared to the terse structural constraints



Scala: Constraint C2 repeated from Tbl. 5.2. We use asScala to convert from Java collections used in the EMF API. The implies and inv functions are implemented in the book's library

fsm.scala/src/main/scala/dsldesign/fsm/scala/constraints.scala

```
val C2 = inv[FiniteStateMachine] { m =>
  m.getStates.asScala.forall { s1 =>
  m.getStates.asScala.forall { s2 =>
```

Python: Very concise thanks to the dedicated comprehension/query syntax. The quantifiers come first, a precondition at the end. Type checking only at runtime. PyEcore helps to use DSLs in

fsm.py/constraints.py with pyecore

```
C2 = lambda m: all ( s1.name != s2.name
  for s1 in m.states for s2 in m.states if s1 != s2)
```

JavaScript: No type checking, not even at runtime; C2 might hold on any object that has 'states' and 'name.' We cast lists to array as the standard list API is too weak. Note the quirky use of the less-than operator as implication, in the "wrong" direction. Ecore.js helps devel server- and

fsm.js/constraints.js with ecore.js

```
var C2 = m =>
  m.get('states').array().every ( s1 =>
  m.get('states').array().every ( s2 =>
    (s1!=s2) <= (s1.get('name')!=s2.get('name')) ))
```

Java: the most verbose of the shown languages; with a bit underdeveloped collection API. We cast lists to streams, in order to access quantifier functions. The constraint could be made more terse using

fsm.java/src/main/java/dsldesign/fsm/java/Constraints.java

```
Function<FiniteStateMachine, Boolean> C2 = m ->
 m.getStates().stream().allMatch ( s1 ->
 m.getStates().stream().allMatch ( s2 ->
```

F#: We show both the LINQ (first) and the functional (second) form for C2. Note that the F# LINQ interface includes a universal quantifier, which makes C2 less cryptic than in C#. The functional formulation suffers from type-impedance be (Seq.toList), like

fsm.fs/Program.fs with .NETModelingFramework

```
let C2: IFiniteStateMachine -> bool = fun m -> query {
  for s1 in m.States do for s2 in m.States do
    where (s1 <> s2) all (s1.Name <> s2.Name) }
let C2a: IFiniteStateMachine -> bool = fun m ->
```

Groovy and **Kotlin** conveniently extend Java collections (using extension methods) with higher-order functions. The default argument "it" in anonymous functions simplifies the constraints slightly. Both examples access the Java API generated by EMF. Kotlin is interesting if your DSL

fsm.groovy/src/main/groovy/dsldesign/fsm/groovy/Constraints.groovy

```
def C2 = {
 it.states.every { s1 ->
 it.states.every { s2 -> s1==s2 || s1.name!=s2.name }}}
```

fsm.kt/src/main/kotlin/dsldesign/fsm/kotlin/Constraints.kt

C#: A Java-like shape of C2 is possible in C#, but we show LINQ syntax to demonstrate a different style, aiming at programmers experienced with database queries.

fsm.cs/Program.cs with .NETModelingFramework

```
Func<IFiniteStateMachine,bool> C2 = m => (
  from s1 in m.States from s2 in m.States
  where s1!=s2 select s1.Name==s2.Name).All (x => !x
```

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

concrete exercises: train building low-level skills

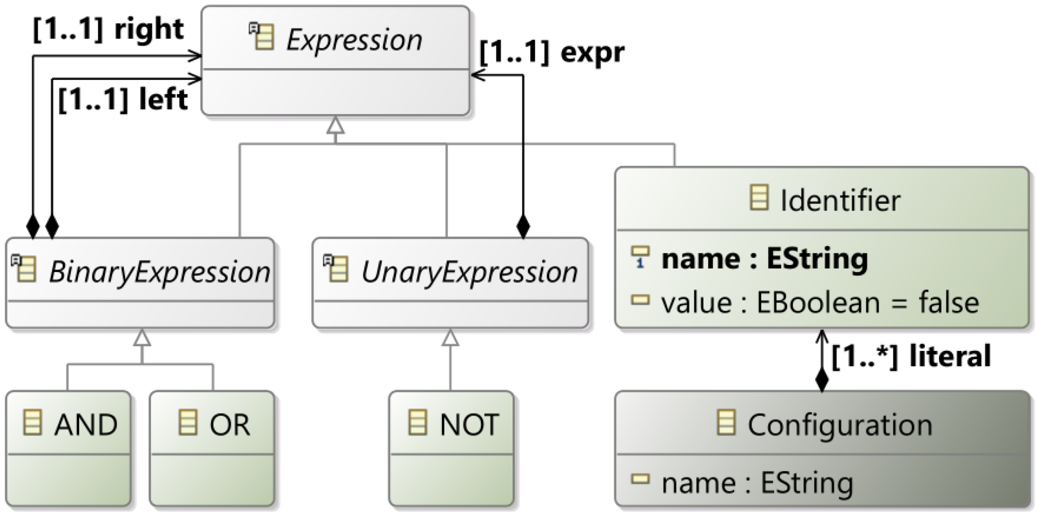
PL perspective linked and mixed with SE perspective

teaching to test is teaching to build

teach small and larger DSLs

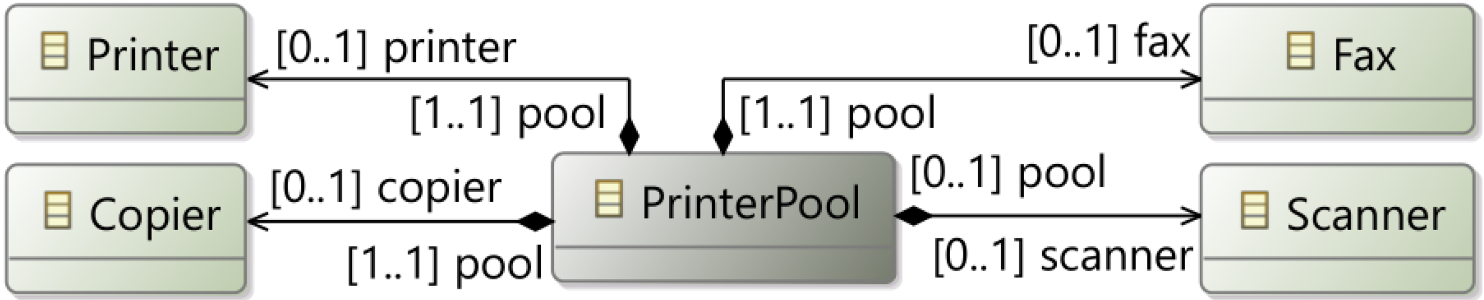
creating instances

Exercise 7.2. Draw an instance of our expression meta-model in Fig. 7.12 for the expression: $\neg((A \wedge \neg B) \vee ((C \wedge \neg D) \vee (\neg C \wedge D)))$. Are different instances possible to represent this expression? In your answer, distinguish between syntactic and semantic equality.



writing constraints

Exercise 5.33. Write the following constraint in the context of the printer pool class in the meta-model: *Each printer pool with a fax must have a printer, and each printer pool with a copier must have a scanner and a printer.*



writing interpreters

interpreter state
(values of variables)

abstract syntax
(an expression in `expr`)

$$\langle\, \varepsilon \,,\, e \,\rangle \rightarrow v$$

value computed by
interpreting e in ε

Exercise 8.7. Add an implication operator $e_1 \rightarrow e_2$ to `expr` (either in Java or Scala), and extend the interpreter accordingly. The result of $e_1 \rightarrow e_2$ is true if and only if e_1 evaluates to false or e_2 evaluates to true. A similar extension can be done for other logical operators: NAND and XOR (Compare with Exercise 7.3 on p. 263.)

Exercise 8.8. Implement the interpreter for `expr` in Python. For simplicity, you can assume that the expressions are parsed and stored in xmi files, so you can load them using pyecore (https://github.com/pyecore/pyecore). In Python, it is natural to use direct recursion (as Ecore generates no switch class for Python). So we recommend to follow the style of our Scala implementation in Fig. 8.7 but using exceptions to raise errors like in our Java implementation.

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

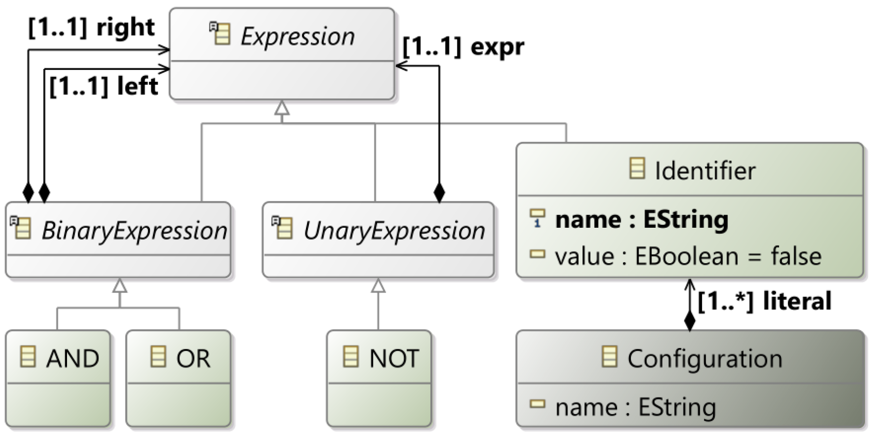
concrete exercises: train building low-level skills

PL perspective linked and mixed with SE perspect

teaching to test is teaching to build

teach small and larger DSLs

meta-models vs. algebraic data types




```scala
sealed abstract class Expression:
  def & (other: Expression): Expression = other match
    case True => this
    case _ => AND (this, other)

  def | (other: Expression): Expression = other match
    case False => this
    case _ => OR (this, other)

  def unary_! : Expression = NOT (this)

sealed abstract class BinaryExpression (
  val left: Expression,
  val right: Expression
) extends Expression

sealed abstract class UnaryExpression (
  val expr: Expression
) extends Expression

case class NOT (e: Expression) extends UnaryExpression (e):
  override def toString = "!" + e

case class AND (l: Expression, r: Expression) extends BinaryExpression (l, r):
  override def toString = "( " + l + " & " + r + " )"

case class OR (l: Expression, r: Expression) extends BinaryExpression (l, r):
  override def toString = "( " + l + " | " + r+ " )"

case class Identifier (name: String ) extends Expression:
  override def toString = name

given StringToExpression: Conversion[String, Identifier] with
  def apply (s: String) = Identifier (s)

case object True extends Expression:
  override def & (other: Expression) = other
  override def unary_! = False
  override def toString = "TRUE"

case object False extends Expression:
  override def | (other: Expression) = other
  override def unary_! = True
  override def toString = "FALSE"

case class Configuration (
  val name: String,
  val identifierValues: Map[Identifier,Expression])
```

model transformation example

FSM to petri nets

Scala: fsm.scala/src/main/scala/dsldesign/fsm/scala/transforms/FsmToPetriNet.scala

```
def convertTransition (places: List[petrinet.Place]) (self: fsm.Transition)
  : petrinet.Transition = pFactory.createTransition before { pnt =>
    pnt.setInput(self.getInput)
    pnt.getFromPlace.addAll (places.filter (_.getName == self.getSource.getName).asJava)
    pnt.getToPlace.addAll (places.filter (_.getName == self.getTarget.getName).asJava) }
```

QVT-O: fsm.qvto/transforms/ToPetriNet.qvto

```
mapping FSM::Transition::ConvertTransition(): PN::Transition{
  input := self.input;
  fromPlace := self.source.resolveone( PN::Place );
  toPlace := self.target.resolveone( PN::Place ); }
```

ATL: fsm.atl/transforms/ToPetriNet.atl

```
rule Transition2Transition{
  from t: FSM!Transition
  to tr: PN!Transition(
    input<-t.input,
    fromPlace<-thisModule.resolveTemp(t.source, 'p'),
    toPlace<-thisModule.resolveTemp(t.target,'p') ) }
```

Scala

QVT-O

ATL

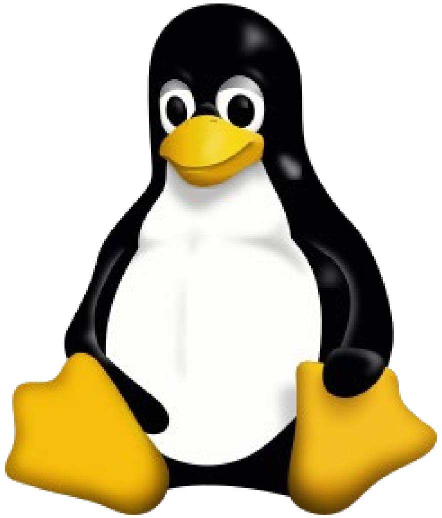
program transformation example

manipulate logical expressions

- constant propagation
- expression simplification
- **conversion into conjunctive normal form (CNF)**
  - using term rewriting
  - specifically strategic programming with kiama https://github.com/inkytonik/kiama

```scala
val demorgansRule = reduce {
  rule[Expression] {
    case NOT (AND (x, y)) => OR (NOT (x), NOT (y))
    case NOT (OR (x, y)) => AND (NOT (x), NOT (y))
  }
}

val doubleNegationRule = reduce {
  rule[Expression] {
    case NOT (NOT (x)) => x
  }
}

val valueNegationRule = everywheretd {
  rule[Expression] {
    case NOT (True) => False
    case NOT (False) => True
  }
}

val distributiveRule = innermost {
  rule[Expression] {
    case OR (x, AND (y, z)) => AND (OR (x, y), OR (x, z))
    case OR (AND (x, y), z) => AND (OR (x, z), OR (y, z))
  }
}

def run (self: Expression): Expression =
  Rewriter.rewrite(
    demorgansRule <*
    doubleNegationRule <*
    valueNegationRule <*
    distributiveRule) (self)
```

if you don't have a transformation language...

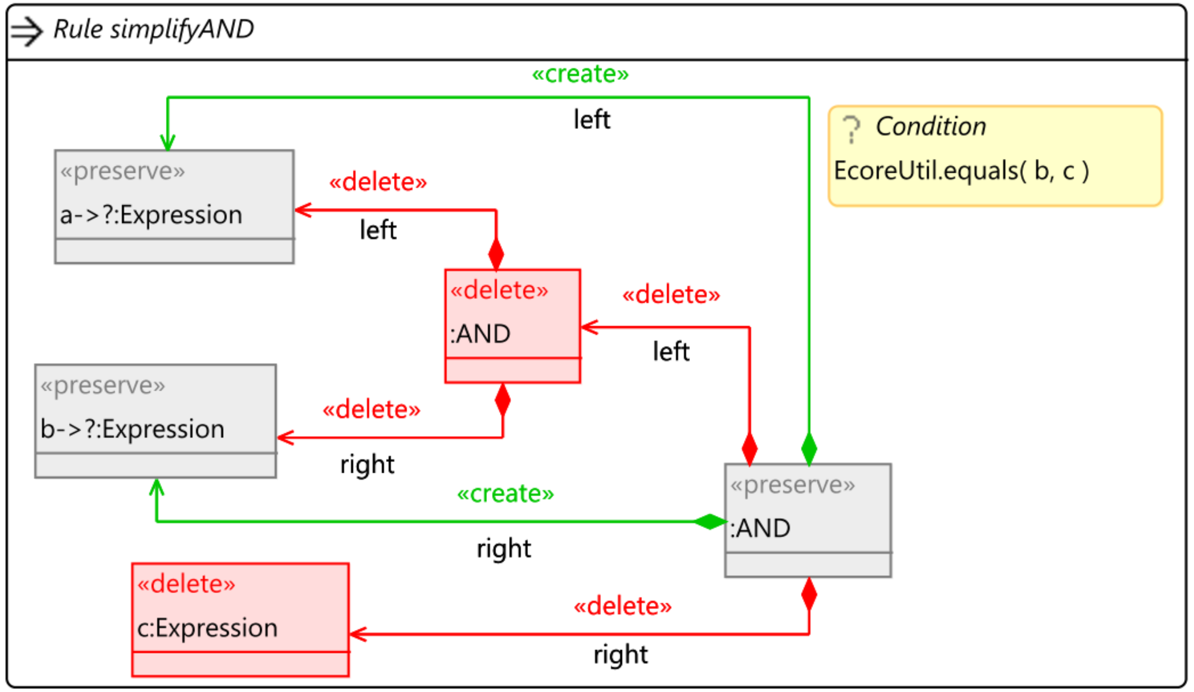


```
/*
 * Recursively performs the following simplifications in-place (as well
 * as the corresponding simplifications with swapped operands):
 *
 * expr && n  ->  n
 * expr && y  ->  expr
 * expr || n  ->  expr
 * expr || y  ->  y
 *
 * Returns the optimized expression.
 */
static struct expr *expr_eliminate_yn(struct expr *e){
   struct expr *tmp;

   if (e) switch (e->type) {
   case E_AND:
      e->left.expr = expr_eliminate_yn(e->left.expr);
      e->right.expr = expr_eliminate_yn(e->right.expr);
      if (e->left.expr->type == E_SYMBOL) {
         if (e->left.expr->left.sym == &symbol_no) {
            expr_free(e->left.expr);
            expr_free(e->right.expr);
            e->type = E_SYMBOL;
            e->left.sym = &symbol_no;
            e->right.expr = NULL;
            return e;
         } else if (e->left.expr->left.sym == &symbol_yes) {
            free(e->left.expr);
            tmp = e->right.expr;
            *e = *(e->right.expr);
            free(tmp);
            return e;
         }
      }
```

or apply the wrong paradigm for your problem?

expression simplification in Henshin (graph transformation)

- rewriting of ((A∧B)∧C) when B =C
- equivalent to one line in Scala



external vs internal DSLs

```
grammar dsldesign.expr.xtext.Expr with org.eclipse.xtext.common.Terminals

import "http://www.dsl.design/dsldesign.expr"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Expression:
    Conjunction ( {OR.left=current} "||" right=Conjunction)* ;

Conjunction returns Expression:
    Unary ( {AND.left=current} "&&" right=Unary)* ;

Unary returns Expression:
    ’(’ Expression ’)’ | {NOT} "!" expr=Unary | Identifier;

Identifier returns Identifier:
    name=EString;

EString returns ecore::EString:
    STRING | ID;
```

external DSL in Xtext
(SE perspective)

```
sealed abstract class Expression:
  def & (other: Expression): Expression = other match
    case True => this
    case _ => AND (this, other)

  def | (other: Expression): Expression = other match
    case False => this
    case _ => OR (this, other)

  def unary_! : Expression = NOT (this)

sealed abstract class BinaryExpression (
  val left: Expression,
  val right: Expression
) extends Expression

sealed abstract class UnaryExpression (
  val expr: Expression
) extends Expression

case class NOT (e: Expression) extends UnaryExpression (e):
  override def toString = "!" + e

case class AND (l: Expression, r: Expression) extends BinaryExpression (l, r):
  override def toString = "( " + l + " & " + r + " )"

case class OR (l: Expression, r: Expression) extends BinaryExpression (l, r):
  override def toString = "( " + l + " | " + r+ " )"

case class Identifier (name: String ) extends Expression:
  override def toString = name

given StringToExpression: Conversion[String, Identifier] with
  def apply (s: String) = Identifier (s)

case object True extends Expression:
  override def & (other: Expression) = other
  override def unary_! = False
  override def toString = "TRUE"
```

internal DSL in Scala
(PL perspective)

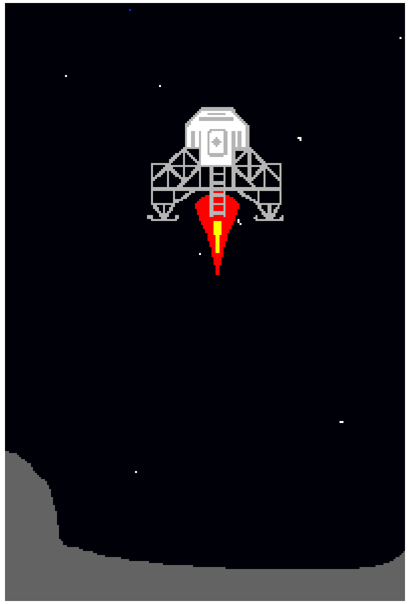
FSM (finite state machine) as an internal DSL

```
val m = (state machine "coffeeMachine"
   initial "initial"
      input "coin"    output "what drink do you want?"     target "selection"
      input "idle"                                         target "initial"
      input "break"   output "machine is broken"           target "deadlock"
   state "selection"
      input "tea"     output "serving tea"                 target "making tea"
      input "coffee"  output "serving coffee"              target "making coffee"
      input "timeout" output "coin returned; insert coin"  target "initial"
      input "break"   output "machine is broken!"          target "deadlock"
   state "making coffee"
      input "done"    output "coffee served. Enjoy!"       target "initial"
      input "break"   output "machine is broken!"          target "deadlock"
   state "making tea"
      input "done"    output "tea served. Enjoy!"          target "initial"
      input "break"   output "machine is broken!"          target "deadlock"
   state "deadlock"
end)
```

Lunar lander

not part of this book ☺

- limitation of our engineering support?



```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 PRINT "Welcome to Baysick Lunar Lander v0.9"
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)

    60 PRINT "You are drifting towards the moon."
    70 PRINT "You must decide how much fuel to burn."
    80 PRINT "To accelerate enter a positive number"
    90 PRINT "To decelerate a negative"

    100 PRINT "Distance " % 'dist % "km, " % "Velocity
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"
    140 GOTO 100
    150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
    160 LET ('fuel := 'fuel - ABS('burn))
    170 LET ('dist := 'dist - 'v)
    180 IF 'dist > 0 THEN 100
    190 PRINT "You have hit the surface"
    200 IF 'v < 3 THEN 240
    210 PRINT "Hit surface too fast (" % 'v % ")km/s"
    220 PRINT "You Crashed!"
    230 GOTO 250
    240 PRINT "Well done"

    250 END

    RUN
  }
}
```

https://www.scala-lang.org/old/node/1403

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

concrete exercises: train building low-level skills

PL perspective linked and mixed with SE perspective

teaching to test is teaching to build

teach small and larger DSLs

testing a program transformation

let's test our transformation of logical expression to conjunctive normal form (CNF)

simple scenarios that cover individual transformation rules

```
class ExpressionToCNFSpec extends
  org.scalatest.freespec.AnyFreeSpec,
  org.scalatest.matchers.should.Matchers:

  val transform = ExpressionToCNF

  "test De Morgan’s rule" in {
    val e = "a" | !("x" & !("y" & "z"))
    val res = rewrite (transform.demorgansRule) (e)
    res should equal ("a" | (!"x" | (!(!"y") & !(!"z"))))
  }

  "test double negation rule" in {
    val e = "a" | !(!"x")
    val res = rewrite (transform.doubleNegationRule) (e)
    res should equal ("a" | "x")
  }

  "test negation of values rule" in {
    val e = "a" & !True | "b" & !False
    val res = rewrite (transform.valueNegationRule) (e)
    res should equal ("a" & False | "b")
  }
```

instance generation

problem: instance generation

- i.e., generate large expressions

the book shows how to use

- Alloy or
- Scalacheck's Gen API

let's implement an instance generator pragmatically

```
def generateExpr (maxNumberOfIdentifiers: Int, maxNestingDepth: Int)
  : Expression =
  val r = Random()
  val identifiers = (26 to (maxNumberOfIdentifiers + 25))
    .map { i =>
      (i % 26 + 65).toChar.toString + (if i/26 == 1 then "" else i/26) }
  subexp (maxNestingDepth, identifiers)

private def subexp (depth: Int, ids:Seq[String]): Expression =
  if depth <= 0
    then Identifier (ids (Random.nextInt (ids.size)))
    else Random.nextInt (4) match
      case 0 => Identifier (ids (Random.nextInt (ids.size)))
      case 1 => NOT (subexp (depth - 1, ids))
      case 2 => AND (subexp (depth - 1, ids), subexp (depth - 1, ids))
      case 3 => OR (subexp (depth - 1, ids), subexp (depth - 1, ids))
```

the oracle problem

test oracle

- tell whether expression is in CNF or not

```scala
class ExpressionToCNFSpec extends
  org.scalatest.freespec.AnyFreeSpec,
  org.scalatest.matchers.should.Matchers:

  val transform = ExpressionToCNF
  ...

  "test 50 randomly generated expressions" in {
    for i <- 1 to 50 do
      val e = generators.generateExpr( 26, 8 )
      isInCNF (transform.run (e)) should be (true)
  }

  /**
   * The idea is to check that in each path to a leaf, there’s no conjunction after
   * a disjunction anymore; and no disjunction or conjunction after a negation.
   */
  def isInCNF (e: Expression): Boolean =
    def checkAllowedNodeTypesInSubtree
      (node: Expression, conjAllowed: Boolean): Boolean = node match
        case OR (l, r) =>
          checkAllowedNodeTypesInSubtree (l, false) &&
          checkAllowedNodeTypesInSubtree (r, false)
        case AND (l, r) => conjAllowed &&
          checkAllowedNodeTypesInSubtree (l, true) &&
          checkAllowedNodeTypesInSubtree (r, true)
        case NOT (Identifier (_)) => true
        case NOT (_) => false
        case _ => true

    checkAllowedNodeTypesInSubtree (e, true)
```

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

concrete exercises: train building low-level skills

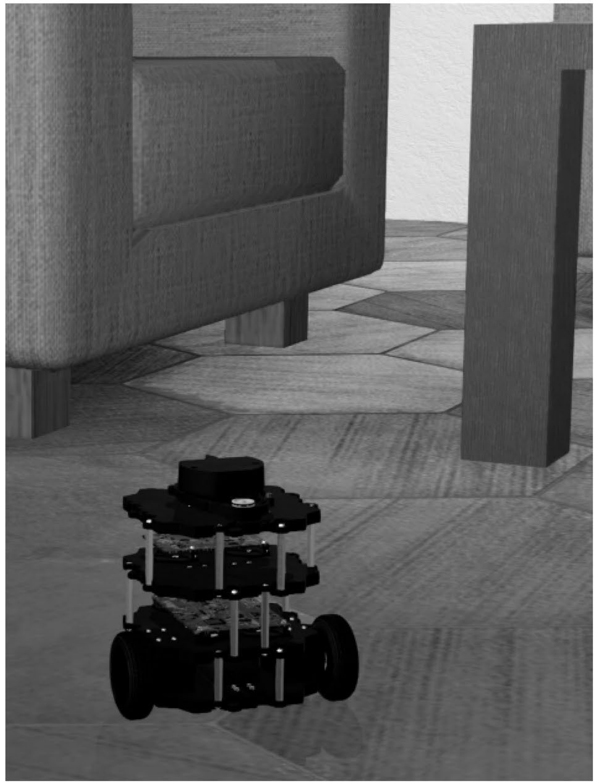
PL perspective linked and mixed with SE perspective

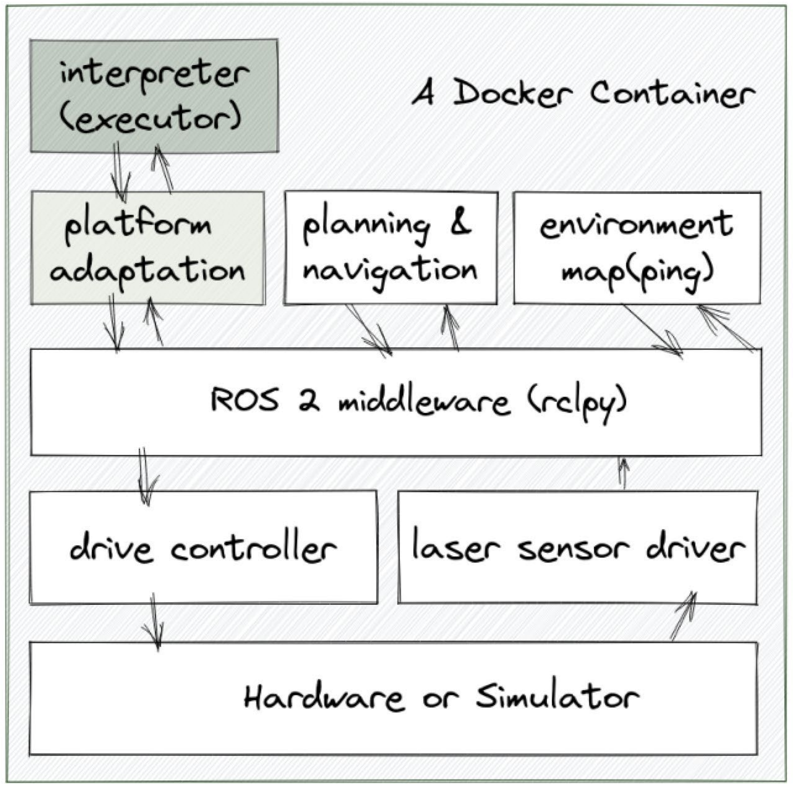
teaching to test is teaching to build

teach small and larger DSLs

larger DSL: robot (with ROS and webots infrastructure)

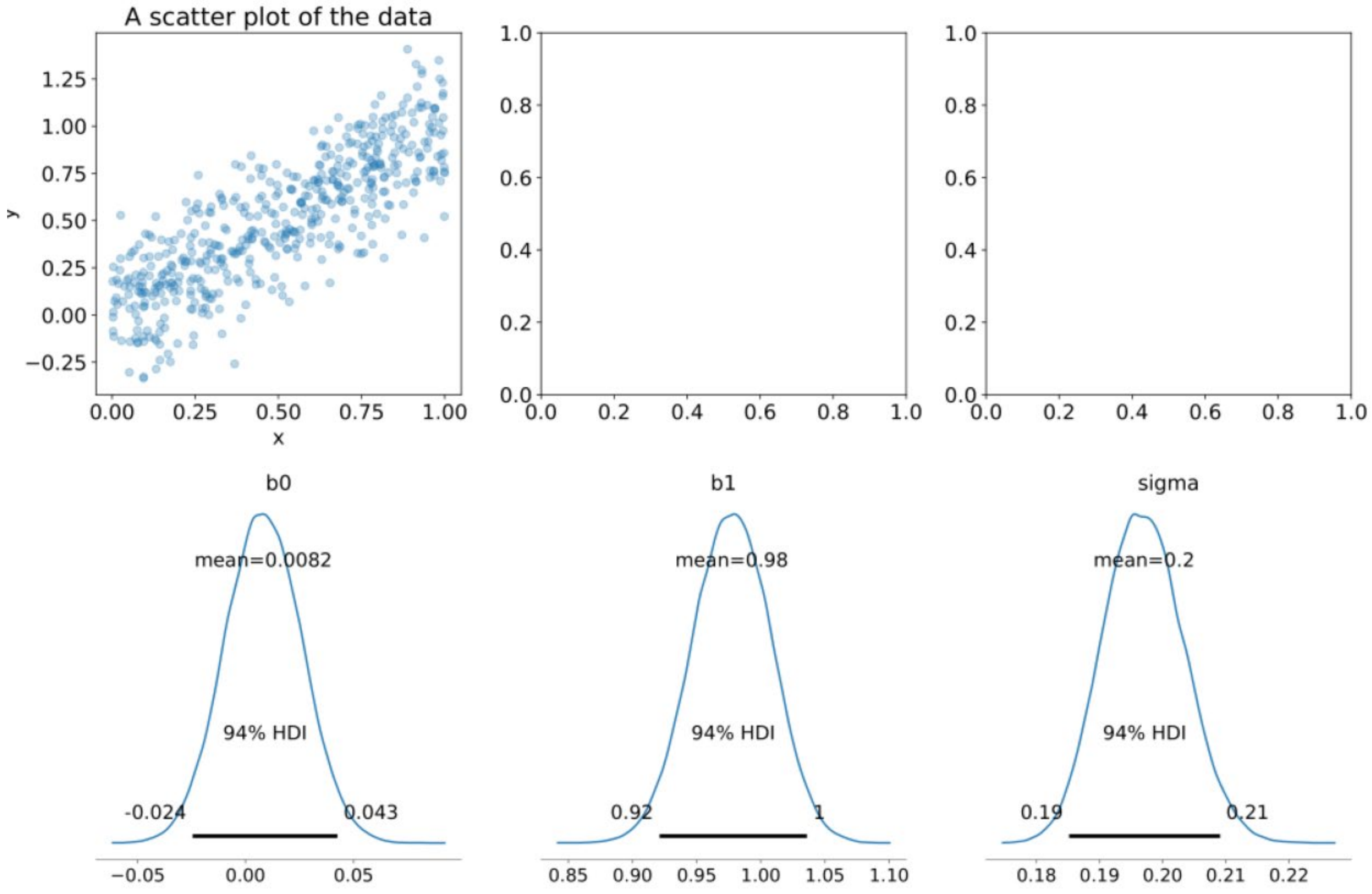
```
-> RandomWalk {
  on clap -> ShutDown

  -> MovingForward {
    move forward at speed 10
    on obstacle -> Avoid
  }

  Avoid {
    move backward for 1 s
    turn by random (-180,180)
  } -> MovingForward

  ShutDown { return to base }
}
```





larger DSL: prpro (probabilistic programming)

with PyMC infrastructure



many more topics in the book

interpretation

code generation

internal DSLs

DSLs for product lines

DSL product lines

concrete examples exemplify principles

definitions: from 'walls of thought' to crisp concepts

concrete exercises: train building low-level skills

PL perspective linked and mixed with SE perspective

teaching to test is teaching to build

teach small and larger DSLs

bringing it all together is challenging

start teaching with an overview example

students find it challenging connecting all the different parts of creating a DSL

Language Component	Purpose	Specification Examples	Example Tools
Concrete syntax Chapter 4	Writing and reading interface for the language: language users write and read programs in concrete syntax.	Regular expressions and context-free grammars.	Parser generators and parsers.
Abstract syntax Chapter 3	An in-memory representation of models and programs as structures in a programming language; a pivotal structure used by the front-end and back-end of the language infrastructure. This is what the language designer uses to implement the language.	Algebraic data types or meta-models.	Produced by parsers, consumed by transformations. Visualized as diagrams or trees for debugging in IDEs.
Static semantics Chapters 5 and 6	Defining valid/invalid models; enforcing well-typedness/constraints impossible/hard to express with grammars and meta-models/ADTs.	First-order constraints, inductive type-system rules, scoping rules.	Advanced frameworks exist, but still mostly implemented manually in practice.
Dynamic semantics Chapters 7 to 9	Define meaning of programs and models; realize the actual purpose of the models.	Code generator or interpreter implemented in a transformation language or in a high-level functional language.	Advanced frameworks exist, but still most languages are implemented manually in practice.
Design environment Chapter 4	Supporting users in creating domain-specific models. The modern editor for your specialized language.	Uses specifications for the other components.	Language workbenches generate high-quality comfortable editors.

exercises, guidelines, examples

277 exercises

71 guidelines

>30 examples
many with sources in our code repository
    http://dsl.design

502 pages

http://dsl.design

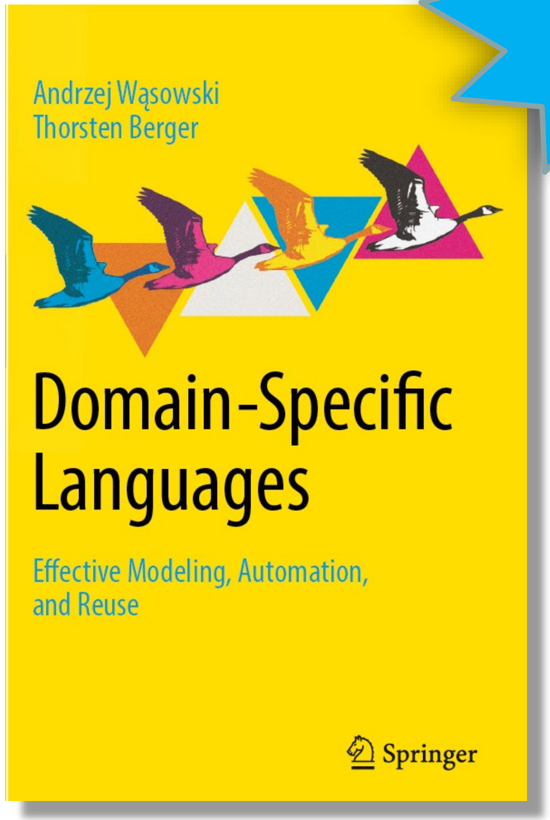
acknowledgements

We thank all students and colleagues who helped to create this book by finding errors, contributing examples, and participating in stimulating discussions. We owe you: **Abdulrashid Masab Mohammed, Alexandru Florin Iosif-Lazar, Anders Fischer Nielsen, Cem Turan, Christoffer Stougaard Pedersen, Daniel Struber, Erik Meijer, Francisco Martinez Lasaca, Georg Hinkel, Hjalte Sorgenfrei Mac Dalland, Jalil Boudjadar, Jean Bezivin, Jean Privat, Jeremy Gibbons, Jonatan Gustafsson, Jurgen Vinju, Karol Wa˛sowski, Kasper Hansen, Marek Furak, Martin Schoeberl, Mietek Bak, Oscar Jonsson, Peter Sestoft, Ralf Gerstner, Ralf Lämmel, Robert Palm, Robin Bellini Olsson, Rolf-Helge Pfeiffer,Stefan Stanciulescu, Sven Peldszus, Tijs van der Storm, Titus Barik, Tobias Schwarz, Vadim Zaytsev, and typeswitch** (of Twitter). Several hundred students in Copenhagen and Gothenburg have bravely taken our courses, being the main motivation for us, while also serving as betatesters. **Phil Watson**, our copy editor at Springer, not only mercilessly pointed out wrong articles, misspellings, and hyphenation errors, but also identified problems in arguments and formulae. We thank you all. All remaining flaws are ours.

Andrzej Wasowski

Thorsten Berger

using the book?
let us know!



http://dsl.design

a new DSL textbook in town!

Thorsten Berger
thorsten.berger@rub.de