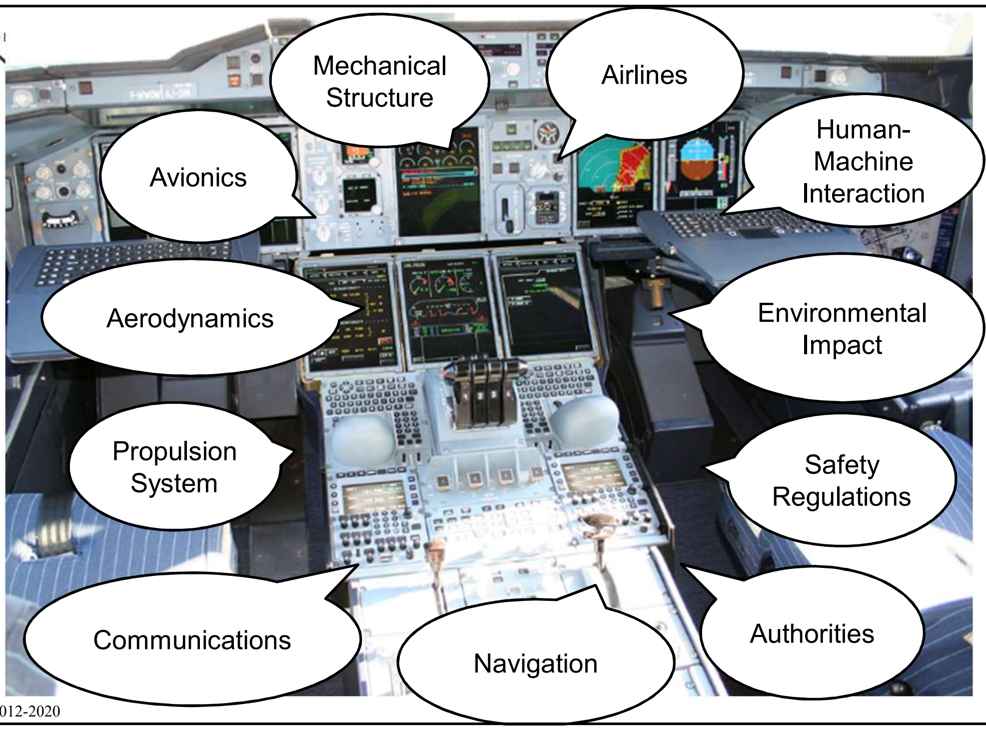
DSL: The Art Of Domain-Specific Languages: Let's Hack Our Own Languages!

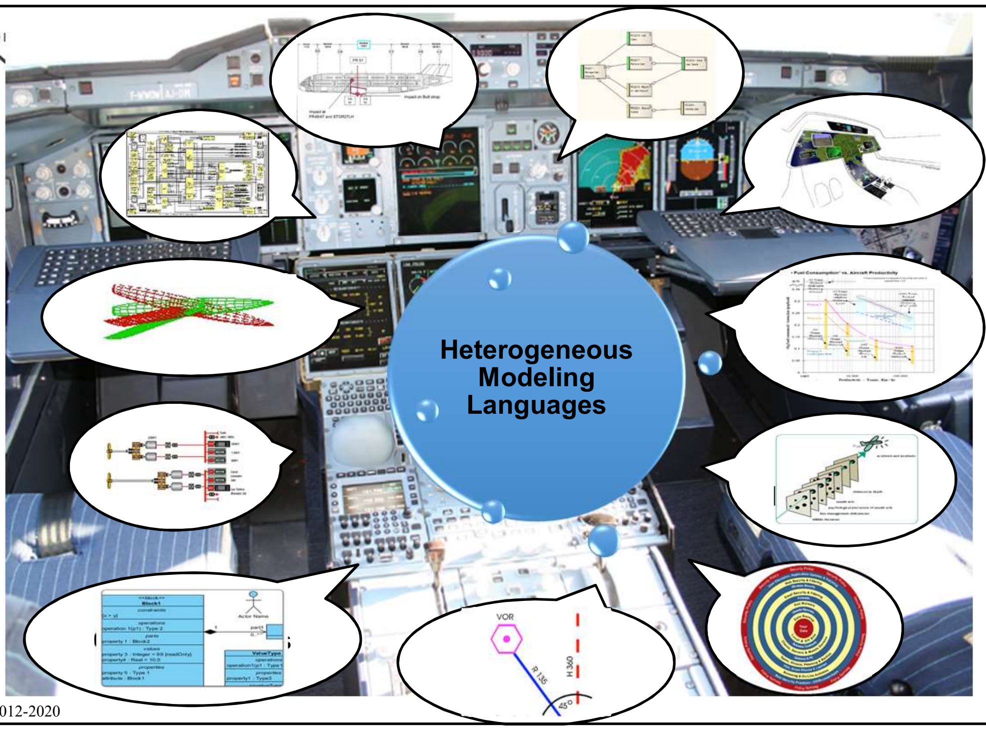
(or: Why I'd like write programs that write programs rather than write programs)

Jean-Marc Jézéquel & Mathieu Acher
e-mail : jezequel@irisa.fr, mathieu.acher@irisa.fr
http://people.irisa.fr/Jean-Marc.Jezequel
Twitter @jmjezequel

Outline

- Introduction to DSL & Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- Operational Semantics with Kermeta
- Building a Compiler: Model transformations
- Conclusion and Wrap-up


Mechanical Structure
Airlines
Avionics
Human-Machine Interaction
Aerodynamics
Environmental Impact
Propulsion System
Safety Regulations
Communications
Navigation
Authorities


Heterogeneous Modeling Languages
VOR

4

UNIVERSITÉ DE RENNES 1

Complex Software Intensive Systems

- Multiple concerns
- Multiple viewpoints & stakeholders
- Multiple domains of expertise
- => Need languages to express them!
  - In a meaningful way for experts
  - With tool support (analysis, code gen., V&V..)
    - Which is still costly to build
  - At some point, all these concerns must be integrated

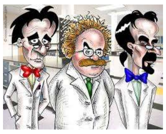
UNIVERSITÉ DE RENNES 1

Example: jHipster

- JHipster is a development platform to generate, develop and deploy Spring Boot + Angular Web applications and Spring microservices.
- **Goal** is to generate a complete and modern Web app or microservice architecture, unifying:
  - A high-performance and robust Java stack on the server side with Spring Boot
  - A sleek, modern, mobile-first front-end with Angular and Bootstrap
  - A robust microservice architecture with JHipster Registry, Netflix OSS, ELK stack and Docker
  - A powerful workflow to build your application with Yeoman, Webpack/Gulp and Maven/Gradle
- **Use of 40+ different DSLs!**

Limits of General Purpose Languages (1)

- **Abstractions** and **notations** used are not natural/suitable for the stakeholders
  - Even with the best languages, impossible to keep all concerns separated down to the implementation



Limits of General Purpose Languages (2)

- Not targeted to a **particular** kind of problem, but to any kinds of software problem.

GLP

DSLs

General-Purpose Languages

« Another lesson we should have learned from the recent past is that the development of 'richer' or 'more powerful' programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally.

I see a great future for very systematic and very modest programming languages »

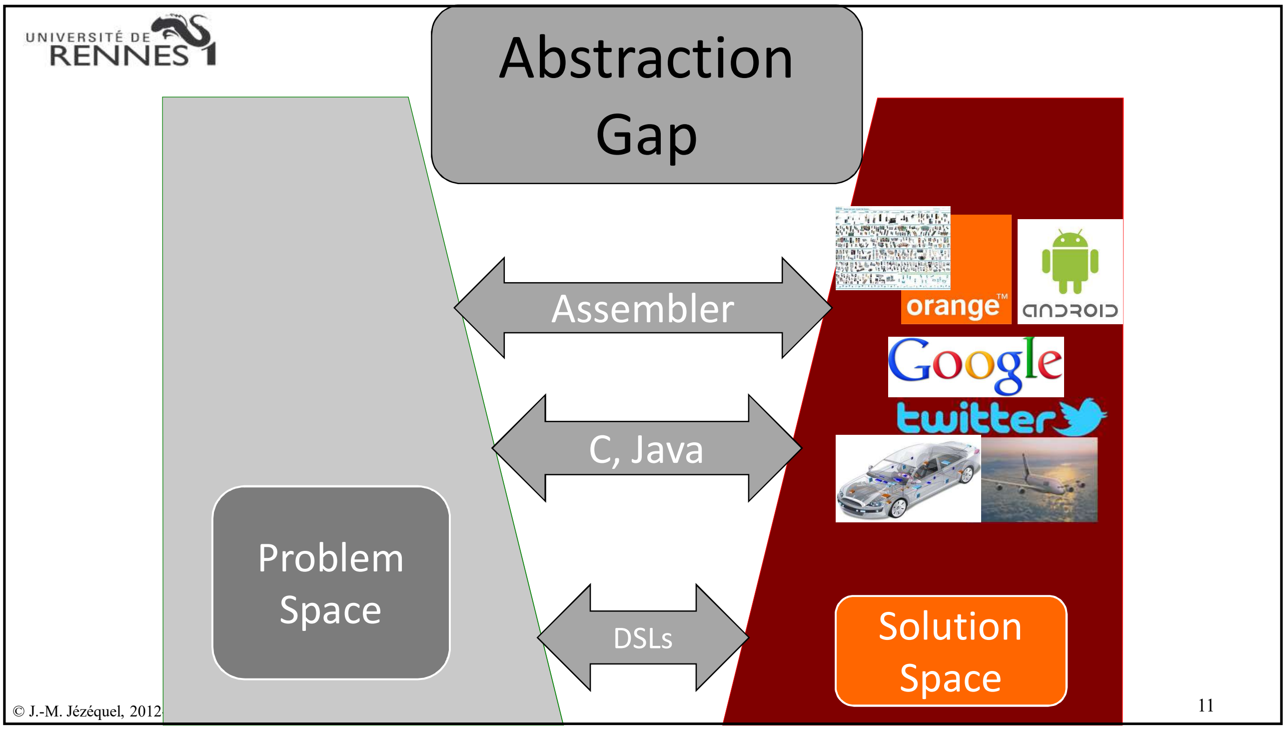
aka **Domain-Specific Languages**

1972

ACM Turing Lecture,
« The Humble Programmer »
Edsger W. Dijkstra

Domain Specific Languages

- Targeted to a **particular** kind of problem
  - with dedicated notations (textual or graphical), support (editor, checkers, etc.)
- Promises: more « efficient » languages for resolving a set of specific problems in a domain
- Each concern described in its own language => reduce abstraction gap



UNIVERSITÉ DE RENNES 1

Evolution towards better abstractions

- The historical approach (50's->80's)
  - Machine = C (M x f) : M x f is « compiled »
    - Typical in Fortran, C, control automation, …
    - Most efficient, but no SoC thus brittle wrt f->f'
- The object oriented revolution (70's -> 2000's)
  - Machine = C(M) x C(f) : M x f is « interpreted » (M still there)
    - Then it makes it easy to have Machine' = C(M) x C(f')
  - Still hard to keep model separated from technical concerns
    - persistency, security, FT, speed…
- One DSL (Domain Specific Language) per concern (90's -> ?)
  - Machine = C(M1) x C(M2) x C(M3) x C(f1) x C(f2) …

UNIVERSITÉ DE RENNES 1

Domain Specific Languages (DSLs)

- Long history: used for almost as long as computing has been done.
- You're using DSLs in a daily basis
  - Even if you do not recognize them as DSLs (yet), because they have many different forms [Fowler]
- **More and more people are building DSLs**
  - **How can we help them?**
  - **In this course, use Model Driven Engineering to *Hack Our Own Languages***

UNIVERSITÉ DE RENNES 1

Why modeling: master complexity

- Modeling, in the broadest sense, is the *cost-effective use of something in place of something else for some cognitive purpose*. It allows us to use something that is *simpler*, *safer* or *cheaper* than reality instead of reality for some purpose.

- A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.

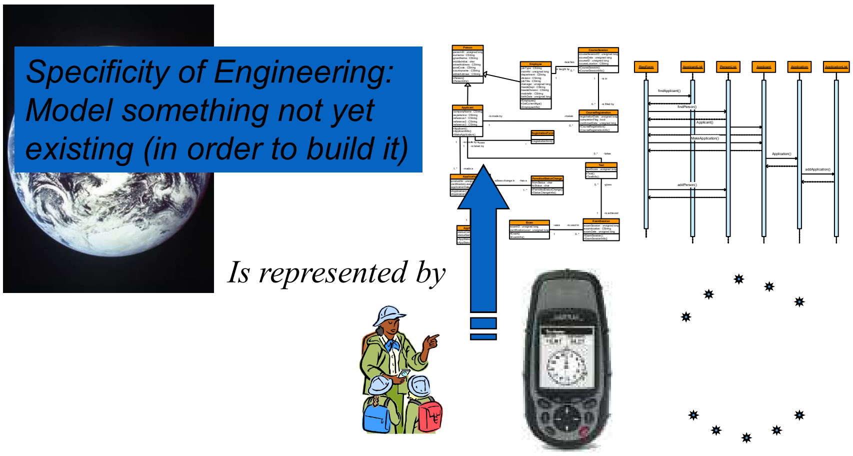
Jeff Rothenberg.

Modeling in Science & Engineering

- A Model is a *simplified* representation of an *aspect* of the World for a specific *purpose*

M_1
(modeling space)

M_0
(the world)

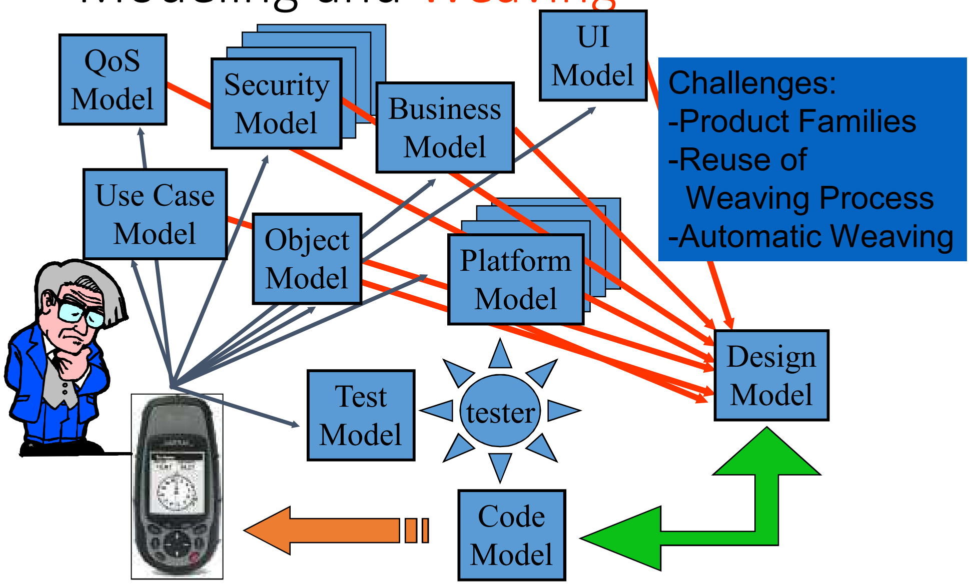


Model and Reality in Software

- Sun Tse: *Do not take the map for the reality*
- Magritte

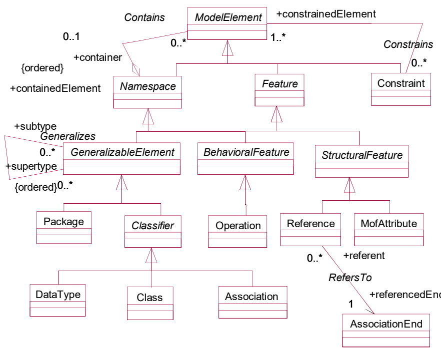
- Software Models: from contemplative to productive

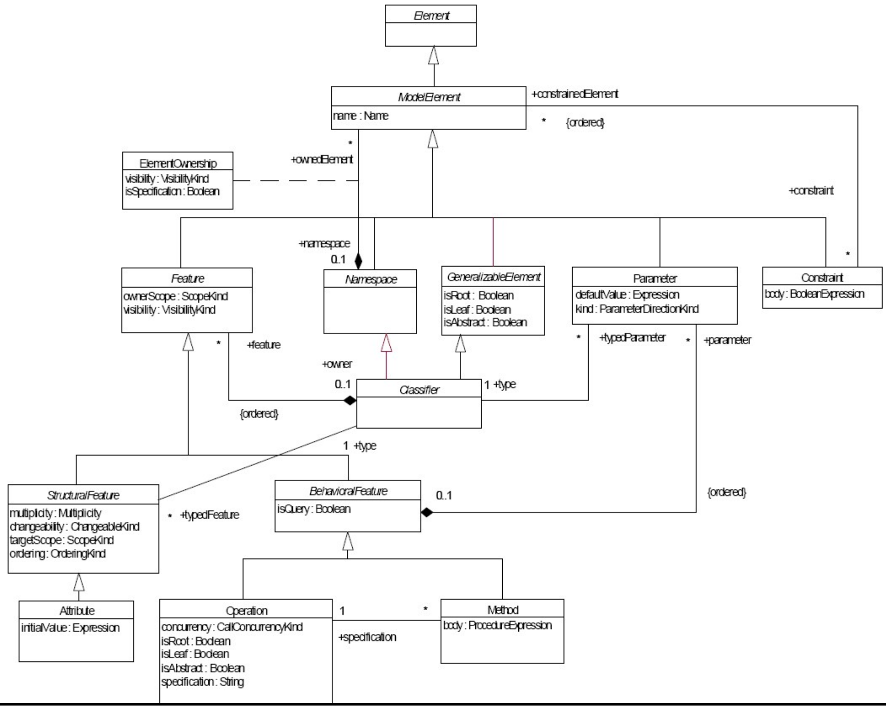
Modeling and Weaving



Assigning Meaning to Models

- If a model *is no longer* just
  - fancy pictures to decorate your room
  - a graphical syntax for C++/Java/C#/Eiffel...
- Then tools must be able to manipulate models
  - Let's make a model of what a model is!
  - => ***meta-modeling***
    - & meta-meta-modeling..
    - Use Meta-Object Facility (MOF) to avoid infinite Meta-recursion





Generalizations

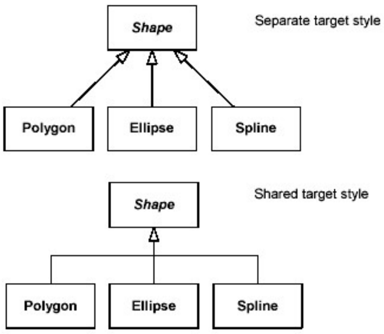


Figure 3-33. Examples of generalizations between classes.

NB: Tell you nothing about:
- *generalization being acyclic,*
- *or semantics of dynamic binding*

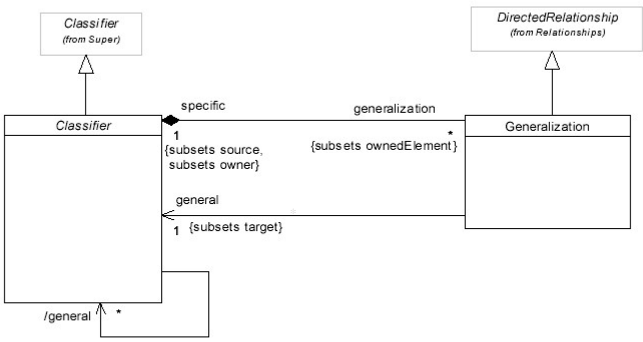
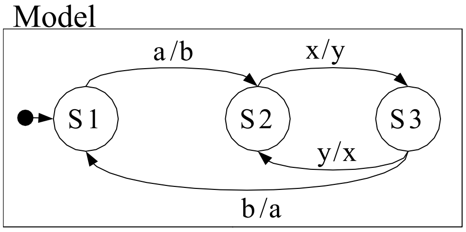
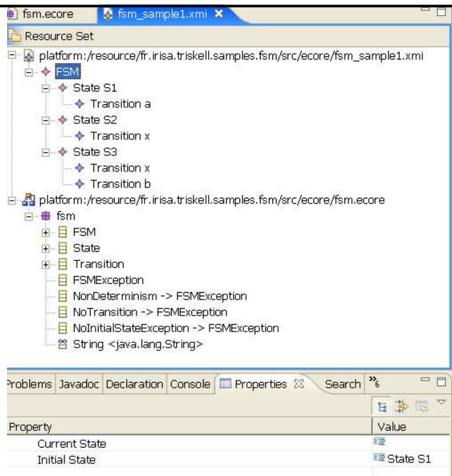


Figure 3-32. The elements defined in the Generalizations package.

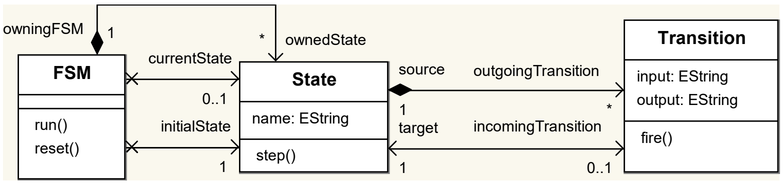
Example with StateMachines

Model





Meta-Model



The 4 layers in practice

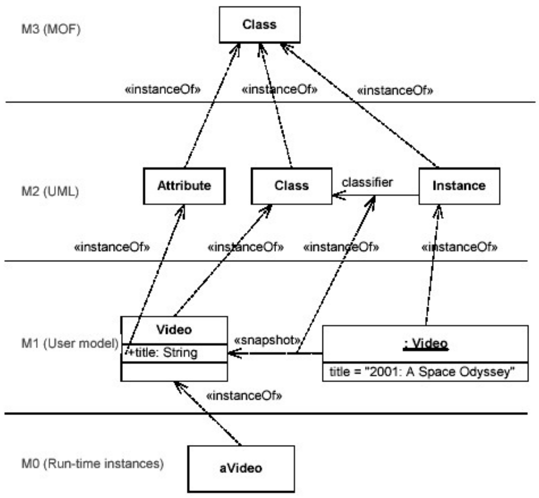
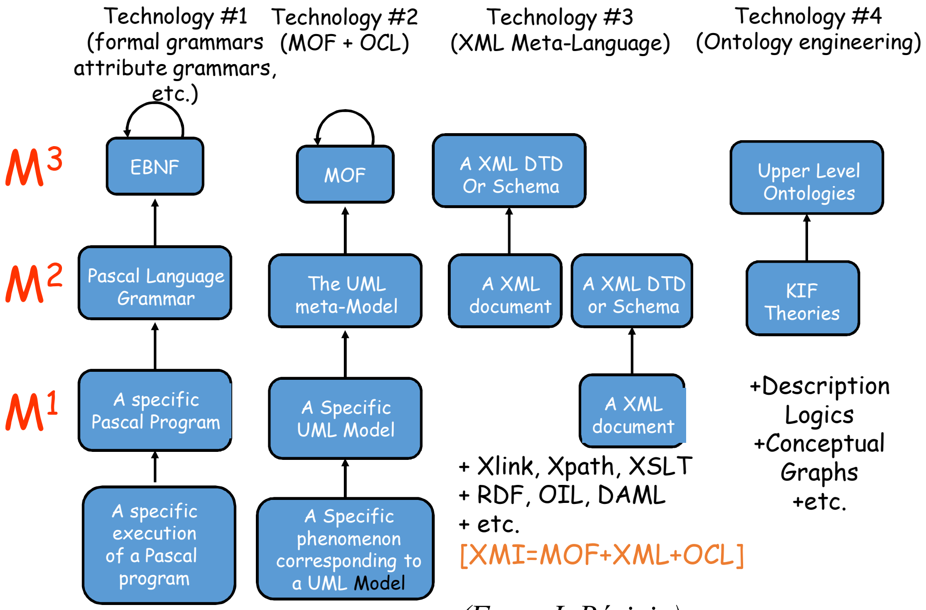


Figure 1-8. An example of the four-layer metamodel hierarchy.

Comparing Abstract Syntax Systems

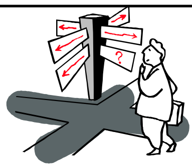


(From J. Bézivin)

Model Driven Engineering : Summary

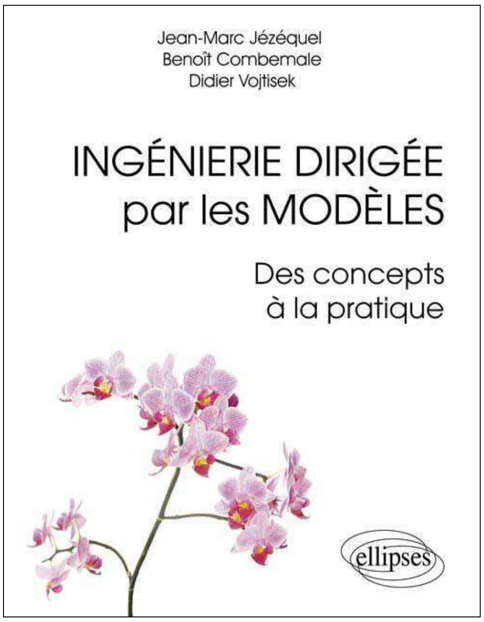
- Modeling to master complexity
  - Multi-dimensional and aspect oriented by definition
- Models: from contemplative to productive
  - Meta-modeling tools, meta-models used to define languages
- Model Driven Engineering
  - Weaving aspects into a design model
    - E.g. Platform Specificities
- Model Driven Architecture (PIM / PSM): just a special case of Aspect Oriented Design
- Related: Generative Prog, Software Factories

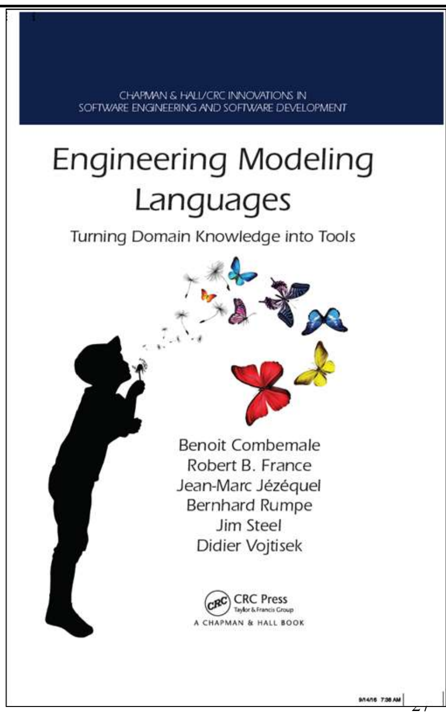
Model Driven Engineering : Summary



- Modeling to master complexity
  - Multi-dimensional and aspect oriented by definition
- Models: from contemplative to productive
  - Meta-modeling tools, meta-models used to define languages
- Model Driven Engineering
  - Weaving aspects into a design model
    - E.g. Platform Specificities
- Model Driven Architecture (PIM / PSM): just a special case of Aspect Oriented Design
- Related: Generative Prog, Software Factories

To learn more…



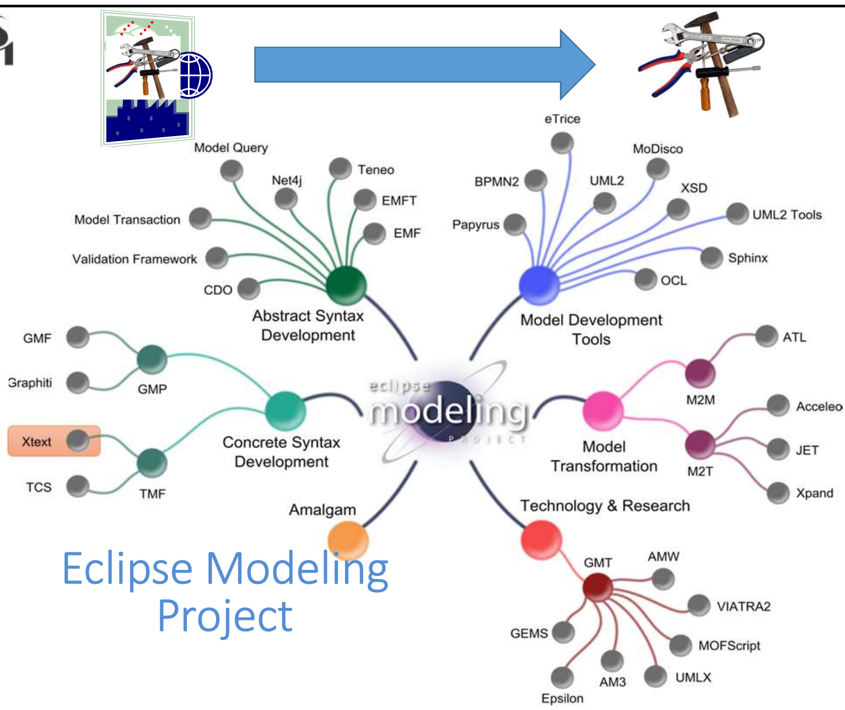


UNIVERSITÉ DE RENNES 1

Outline

- Introduction to Model Driven Engineering
- **Designing Meta-models: the LOGO example**
- Static Semantics with OCL
- Operational Semantics with Kermeta
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

UNIVERSITÉ DE RENNES 1



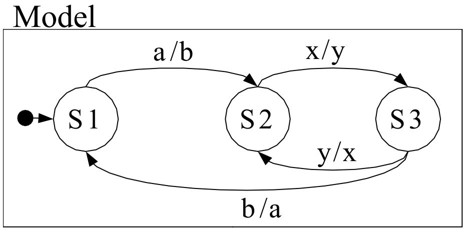
Eclipse Modeling Project

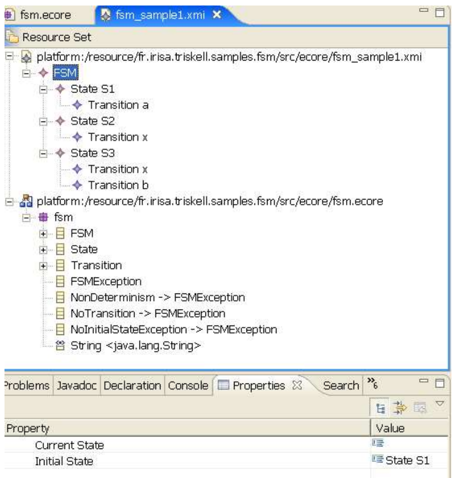
Meta-Models as Shared Knowledge

- Definition of an Abstract Syntax in E-MOF
  - Repository of models with EMF
  - Reflexive Editor in Eclipse
  - JMI for accessing models from Java
  - XML serialization for model exchanges
- Applied in more and more projects
  - SPEEDS, OpenEmbedd,DiVA…

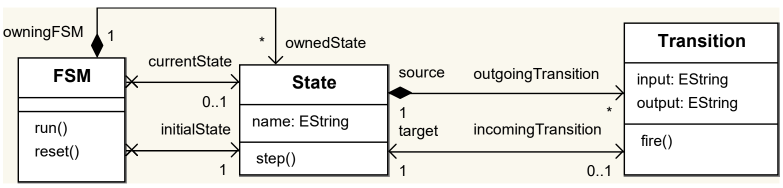
Example with StateMachines

Model

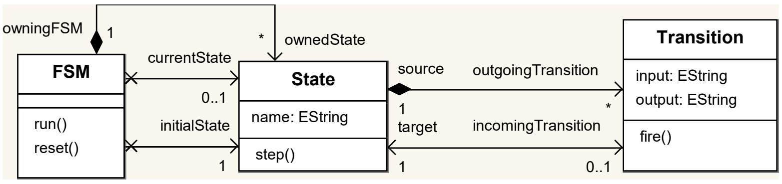




Meta-Model



Breathing life into Meta-Models



// MyKermetaProgram.kmt
// An E-MOF metamodel is an OO program that does nothing

require "StateMachine.ecore" *// to import it in Kermeta*

// Kermeta lets you weave in **aspects**

// Contracts (OCL WFR)

require "StaticSemantics.ocl"

// Method bodies (Dynamic semantics)

require "DynamicSemantics.xtend"

// Transformations

```
Context FSM
inv: ownedState->forAll(s1,s2|
s1.name=s2.name implies s1=s2)
```

```
class FSM {
   public def void reset()  {
         currentState = initialState
```

```
class Minimizer {
   public def FSM minimize (source: FSM) {…}
}
```


Tools built with MDE

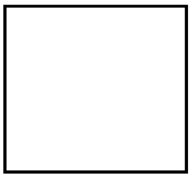
- A tool (aka Model Transformation) is just a program working with specific OO data structures (aka meta-models) representing abstract syntax ~~trees~~ (graphes).
  - Kermeta approach: organize the program along the OO structure of the meta-model
  - Any software engineer can now build a DSL toolset!
    - No longer just for genius…
- Product Lines of DSLs = SPL of OO programs
  - Safe reuse of the tool chains -> Static typing
  - Backward compatibility, Migration of artifacts -> Adaption

DIY with LOGO programs

- Consider LOGO programs of the form:

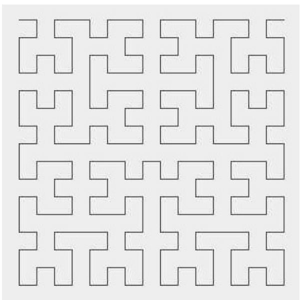
```
repeat 3  [ pendown forward 3 penup forward 4  ]
```

```
to square :width
  repeat 4  [ forward :width right  90]
end
pendown square 10 *10
```



Fractals in LOGO

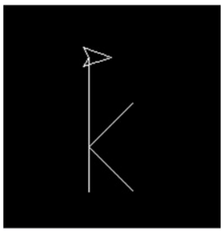
```
; lefthilbert
to lefthilbert :level :size
  if :level != 0 [
    left 90
    righthilbert :level-1 :size
    forward :size
    right 90
    lefthilbert :level-1 :size
    forward :size
    lefthilbert :level-1 :size
    right 90
    forward :size
    righthilbert :level-1 :size
    left 90
  ]
end
```

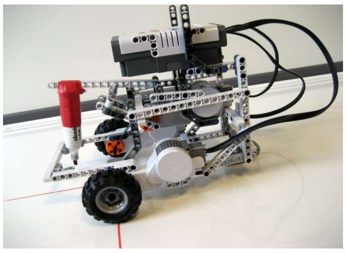


```
; righthilbert
to righthilbert :level :size
  if :level != 0 [
    right 90
    lefthilbert :level-1 :size
    forward :size
    left 90
    righthilbert level-1 :size
    forward :size
    righthilbert :level-1 :size
    left 90
    forward :size
    lefthilbert :level-1 :size
    right 90
  ]
end
```

Case Study: Building a Programming Environment for Logo

- Featuring
  - Edition in Eclipse
  - On screen simulation
  - Compilation for a Lego Mindstorms robot





Model Driven Language Engineering : the Process

- Specify abstract syntax
- Specify concrete syntax
- Build specific editors
- Specify static semantics
- Specify dynamic semantics
- Build simulator
- Compile to a specific platform

Meta-Modeling LOGO programs

- Let's build a meta-model for LOGO
  - Concentrate on the abstract syntax
  - Look for concepts: instructions, expressions…
  - Find relationships between these concepts
    - It's like UML modeling !

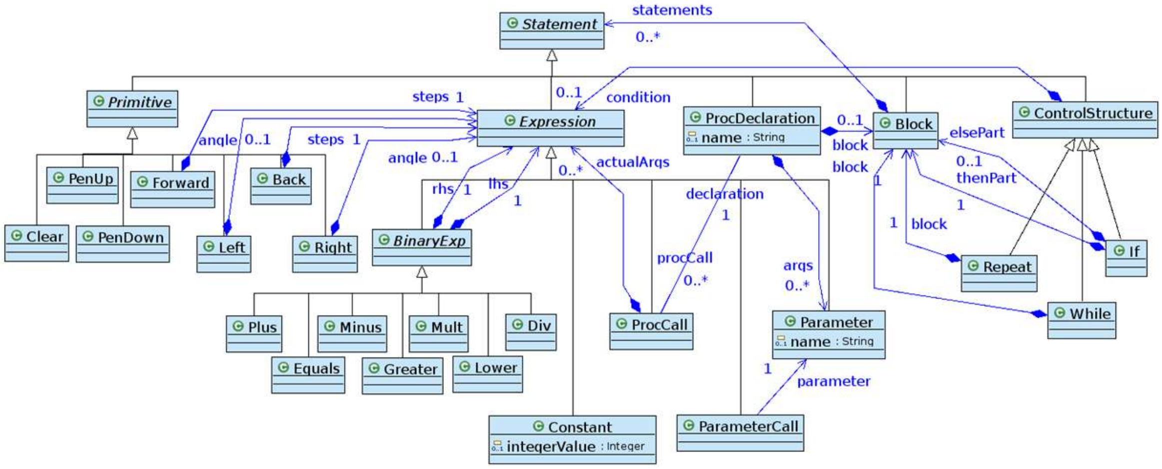
- Defined as an ECore model
  - Using EMF tools and editors

LOGO metamodel : find the program concepts

```
; lefthilbert
to lefthilbert :level :size
  if :level != 0 [
    left 90
    righthilbert :level-1 :size
    forward :size
    right 90
    lefthilbert :level-1 :size
    forward :size
    lefthilbert :level-1 :size
    right 90
    forward :size
    righthilbert :level-1 :size
    left 90
  ]
```

- *Comment (?)*
- *ProcDeclaration, Parameter (formal)*
- *If*
- *ParameterCall, NotEqual, Constant (BinaryExp, Expression)*
- *Block*
- *Left (Primitive Instruction), Constant*
- *ProcedureCall, Expression, Expression*
- *…*

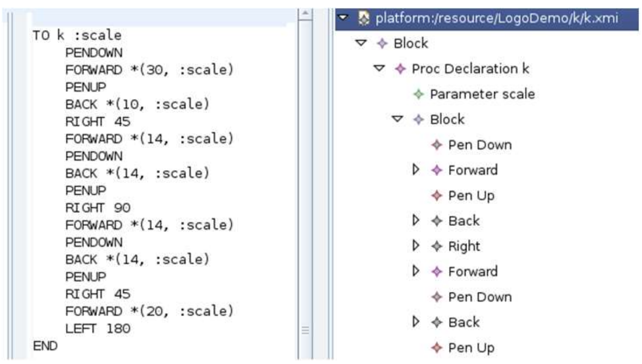
LOGO metamodel



ASMLogo.ecore

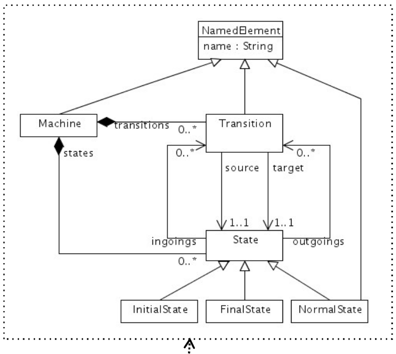
Concrete syntax

- Any regular EMF based tools
- Textual using Xtext
- Graphical using GMF



Grammar ↔ xtext ↔ MetaModel

```
machineDefinition:
  MACHINE OPEN_SEP stateList
  transitionList CLOSE_SEP;

stateList:
  state (COMMA state)*;

state:
  ID_STATE;

transitionList:
  transition (COMMA transition)*;

transition:
  ID_TRANSITION OPEN_SEP
  state state CLOSE_SEP;

MACHINE: 'machine';
OPEN_SEP: '{';
CLOSE_SEP: '{';
COMMA: ',';
ID_STATE: 'S' ID;
ID_TRANSITION: 'T' (0..9)+;
ID: (a..zA..Z_) (a..zA..Z0..9)*;
```



conforms To

conforms To

```
machine {
  SOne STwo
  T1 { SOne STwo }
}
```

Source Code/Model

Outline

- Introduction to Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- Operational Semantics with Kermeta
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

Static Semantics with OCL

- Complementing a meta-model with Well-Formedness Rules, aka *Contracts* e.g.;
  - A procedure is called with the same number of arguments as specified in its declaration
- Expressed with the OCL (Object Constraint Language)
  - The OCL is a language of typed expressions.
  - A constraint is a valid OCL expression of type Boolean.
  - A constraint is a restriction on one or more values of (part of) an object-oriented model or system.

Contracts in OO languages

- Inspired by the notion of Abstract Data Type
- Specification = Signature +
  - Preconditions
  - Postconditions
  - Class Invariants
- Behavioral contracts are inherited in subclasses

OCL

- Can be used at both
  - M1 level (constraints on Models)
    - aka *Design-by-Contract* (Meyer)
  - M2 level (constraints on Meta-Models)
    - aka Static semantics
- Let's overview it with M1 level exemples

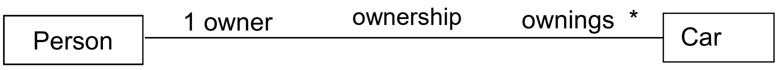
Simple constraints

Customer
name: String title: String age: Integer isMale: Boolean

```
title = if isMale then 'Mr.' else 'Ms.' endif
age >= 18 and age < 66
name.size < 100
```

Non-local contracts: navigating associations

- Each association is a navigation path
  - The context of an OCL expression is the starting point
  - Role names are used to select which association is to be traversed (or target class name if only one)



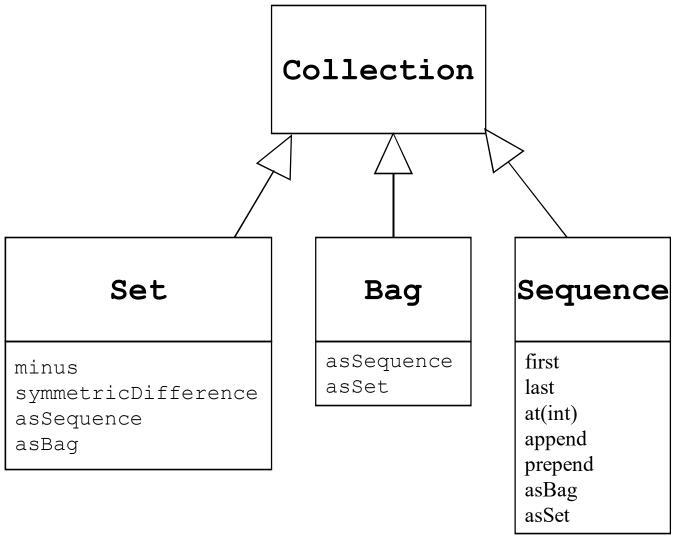
Context Car inv:
self.owner.age >= 18

Navigation of 0..* associations

- Through navigation, we no longer get a scalar but a *collection* of objects
- OCL defines 3 sub-types of collection
  - **Set** : when navigation of a 0..* association
    - *Context Person inv: ownings* return a Set[Car]
    - Each element is in the Set at most once
  - **Bag :** if more than one navigation step
    - An element can be present more than once in the Bag
  - **Sequence** : navigation of an association {ordered}
    - It is an ordered Bag
- Many predefined operations on type *collection*

Syntax::
Collection->operation

Collection hierarchy



Basic operations on collections

- *isEmpty*
  - *true* if collection has no element

Context Person inv:
age<18 implies ownings->isEmpty

- *notEmpty*
  - *true* if collection has at least one element
- *size*
  - Number of elements in the collection
- *count (elem)*
  - Number of occurrences of element *elem* in the collection

select Operation

- possible syntax
  - collection->select(elem:T | expr)
  - collection->select(elem | expr)
  - collection->select(expr)
- Selects the subset of *collection* for which property *expr* holds
- e.g.

```
context Person inv:
ownings->select(v: Car | v.mileage<100000)->notEmpty
```

- shortcut:

```
context Person inv:
ownings->select(mileage<100000)->notEmpty
```


forAll Operation

- possible syntax
  - collection->forall(elem:T | expr)
  - collection->forall(elem | expr)
  - collection->forall(expr)
- True iff *expr* holds for each element of the *collection*
- e.g.

```
context Person inv:
ownings->forall(v: Car | v.mileage<100000)
```

- shortcut:

```
context Person inv:
ownings->forall(mileage<100000)
```

Operations on Collections

Operation	Description
size	The number of elements in the collection
count(object)	The number of occurences of object in the collection.
includes(object)	True if the object is an element of the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
isEmpty	True if the collection contains no elements.
notEmpty	True if the collection contains one or more elements.
iterate(expression)	Expression is evaluated for every element in the collection.
sum(collection)	The addition of all elements in the collection.
exists(expression)	True if expression is true for at least one element in the collection.
forAll(expression)	True if expression is true for all elements.

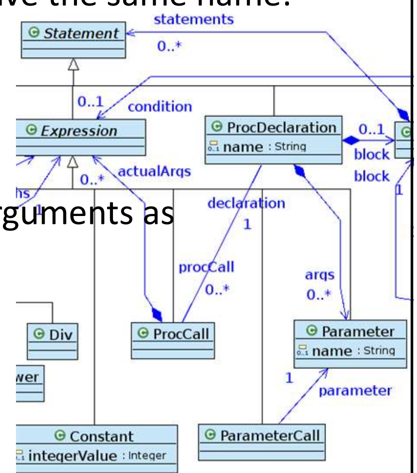
Static Semantics for LOGO

- No two formal parameters of a procedure may have the same name:

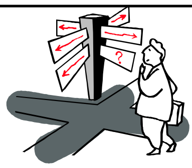
```
context ProcDeclaration
  inv unique_names_for_formal_arguments :
    args -> forAll ( a1 , a2 | a1. name = a2.name
             implies a1 = a2 )
```

- A procedure is called with the same number of arguments as specified in its declaration:

```
context ProcCall
  inv same_number_of_formals_and_actuals :
    actualArgs -> size = declaration .args -> size
```



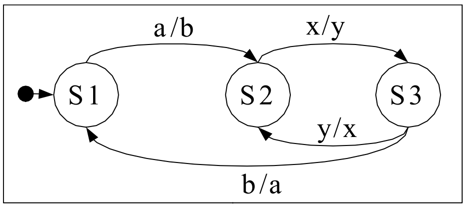
Outline



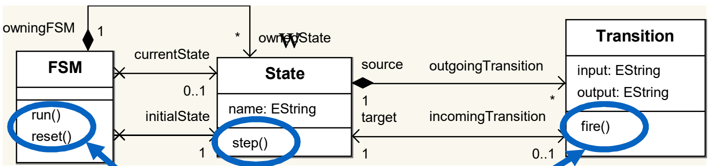
- Introduction to Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- **Operational Semantics with Kermeta**
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

Operational Semantics of State Machines

- A model



- Its metamodel

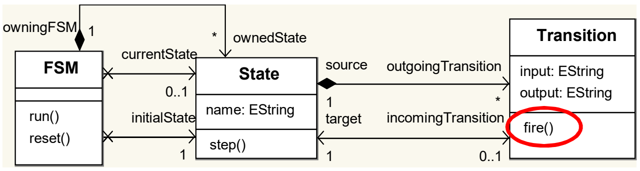


- Adding Operational Semantics to OO Metamodels

Kermeta Action Language: XTEND

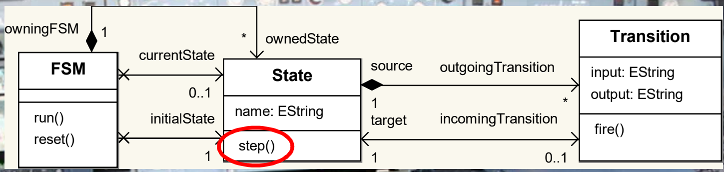
- Xtend
  - flexible and expressive dialect of Java
  - compiles into readable Java 5 compatible source code
  - can use any existing Java library seamlessly
- Among features on top of Java:
  - Extension methods
    - enhance closed types with new functionality
  - Lambda Expressions
    - concise syntax for anonymous function literals (like in OCL)
  - ActiveAnnotations
    - annotation processing on steroids
  - Properties
    - shorthands for accessing & defining getters and setter (like EMF)

Example with Xtend

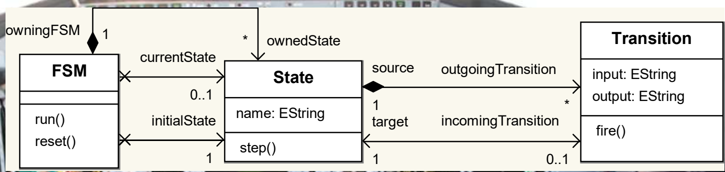


public def String fire()

```
_self.source.owningFSM.currentState = _self.target
return _self.output
```




```
def String step(String c) {
  // Get the valid transitions
  var validTransitions =
      _self.outgoingTransition.filter[t|t.input.equals(c)]

  // Check if there is one and only one valid transition
  if(validTransitions.empty) throw new NoTransition
  if(validTransitions.size > 1) throw new NonDeterminism

  // Fire the transition
  return validTransitions.get(0).fire()
}
```

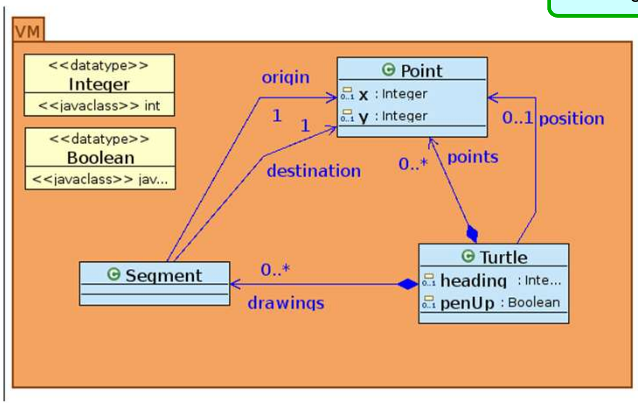

```
def void run() {
// reset if there is no current state
if (_self.currentState == null) _self.currentState = _self.initialState
var str = ""
while (str != "quit") {
  println("Current state : " + _self.currentState.name)
  str = Console.instance.readLine("give me a letter : ")
  try {
    var textRes = _self.currentState.step(str)
    if (textRes == void || textRes == "") textRes = "NC"
    println("string produced : " + textRes)
  } catch (NonDeterminism err) {
    println(err.toString)
    str = "quit"
  } catch (NoTransition err) {
    println(err.toString)
    str = "quit"
  }
}
```

Operational Semantics for LOGO

- Expressed as a mapping from a meta-model to a virtual machine (VM)
- LOGO VM ?
  - Concept of Turtle, Lines, points…
  - Let's Model it !
  - (Defined as an Ecore meta-model)

Virtual Machine - Model

VMLogo.ecore



- Defined as an Ecore meta-model

Virtual Machine - Semantics

LogoVMSemantics.kmt

```
require "VMLogo.ecore"
require "TurtleGUI.kmt"

aspect class Point {
  def String toString() {
    return "[" + x.toString + "," + y.toString + "]"
  }
}

aspect class Turtle {
  def void setPenUp(b : Boolean) {
    penUp = b
  }
  def void rotate(angle : Integer)  {
    heading = (heading + angle).mod(360)
  }
}
```


Map Instructions to VM Actions

- Weave an interpretation aspect into the meta-model
  - add an *eval()* method into each class of the LOGO MM

```
aspect class PenUp {
        def int eval (ctx: Context) {

          ctx.getTurtle().setPenUp(true)
     }
…
aspect class Clear {
        def int eval (ctx: Context) {
                ctx.getTurtle().reset()
     }
```

Meta-level Anchoring

- Simple approach using the Kermeta VM to « ground » the semantics of basic operations
- Or reify it into the LOGO VM
  - Using eg a stack-based machine
  - Ultimately grounding it in kermeta though

```
…
aspect class Add {
   def int eval (ctx: Context)  {
          return lhs.eval(ctx)
         + rhs.eval(ctx)
}
```

```
…
aspect class Add {
   def void eval (ctx: Context) {
          lhs.eval(ctx) // put result
         // on top of ctx stack
          rhs.eval(ctx) // idem
          ctx.getMachine().add()
}
```


Handling control structures

- Block
- Conditional
- Repeat
- While

Operational semantics

LogoDynSemantics.kmt

```
require "ASMLogo.ecore"
require "LogoVMSemantics.kmt"

aspect class If {
  def int eval(context : Context) {
    if (condition.eval(context) != 0)
      return thenPart.eval(context)
    else return elsePart.eval(context)
  }
}

aspect class Right {
  def int eval(context : Context) {
    return context.turtle.rotate(angle.eval(context))
  }
}
```


Handling function calls

- Use a stack frame
  - Owned in the Context
- Bind formal parameters to actual
- Push stack frame
- Execute method body
- Pop stack frame

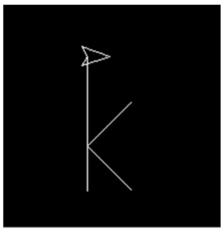
Getting an Interpreter

- Glue that is needed to load models
  - ie LOGO programs
- Vizualize the result
  - Print traces as text
  - Put an observer on the LOGO VM to graphically display the resulting figure

Simulator

- Execute the operational semantics

```
TO k :scale
    PENDOWN
    FORWARD *(30, :scale)
    PENUP
    BACK *(10, :scale)
    RIGHT 45
    FORWARD *(14, :scale)
    PENDOWN
    BACK *(14, :scale)
    PENUP
    RIGHT 90
    FORWARD *(14, :scale)
    PENDOWN
    BACK *(14, :scale)
    PENUP
    RIGHT 45
    FORWARD *(20, :scale)
    LEFT 180
END

CLEAR
$k(4)
```



```
Problems | Javadoc | Declaration | Console ⊠ | Pro
KM Logo Console
Launching logo interpreter on file : /home/
Tortue trace vers [0,120]
Tortue se deplace en [0,80]
Tortue se deplace en [39,119]
Tortue trace vers [0,80]
Tortue se deplace en [39,41]
Tortue trace vers [0,80]
Tortue se deplace en [0,0]
Execution terminated successfully.
```

UNIVERSITÉ DE RENNES 1

Outline

- Introduction to Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- Operational Semantics with Kermeta
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

UNIVERSITÉ DE RENNES 1

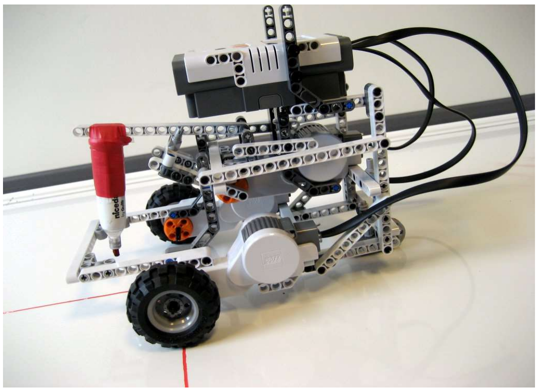
Implementing a model-driven compiler

- Map a LOGO program to Lego Mindstroms
  - The LOGO program is like a PIM
  - The target program is a PSM
  - => model transformation
- Kermeta to weave a « compilation » aspect into the logo meta-model

```
aspect class PenUp {
        def void compile (ctx: Context) {

    }
…
aspect class Clear {
    }
```

Specific platform

- Lego Mindstorms Turtle Robot
  - Two motors for wheels
  - One motor to control the pen



Model-to-Text vs. Model-to-Model

- Model-to-Text Transformations
  - For generating: code, xml, html, doc.
  - Should be limited to syntactic level transcoding
- Model-to-Model Transformations
  - To handle more complex, semantic driven transformations
    - PIM to PSM a la OMG MDA
    - Refining models
    - Reverse engineering (code to models)
    - Generating new views
    - Applying design patterns
    - Refactoring models
    - Deriving products in a product line
    - … any model engineering activity that can be automated…

Model-to-Text Approaches

- For generating: code, xml, html, doc.
  - Visitor-Based Approaches:
    - Some visitor mechanisms to traverse the internal representation of a model and write code to a text stream
    - Iterators, Write ()
  - Template-Based Approaches
    - A template consists of the target text containing slices of meta-code to access information from the source and to perform text selection and iterative expansion
    - The structure of a template resembles closely the text to be generated
    - Textual templates are independent of the target language and simplify the generation of any textual artefacts

Model to Text in practice

- For simple cases, use the template mecanism of Xtend
  - Output = ``` template expression’’’
- Many template generators for MDE do exist
  - E.g. Acceleo (from Obeo) is quite popular in industry
    - a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard
    - http://www.eclipse.org/acceleo/

Example with Acceleo

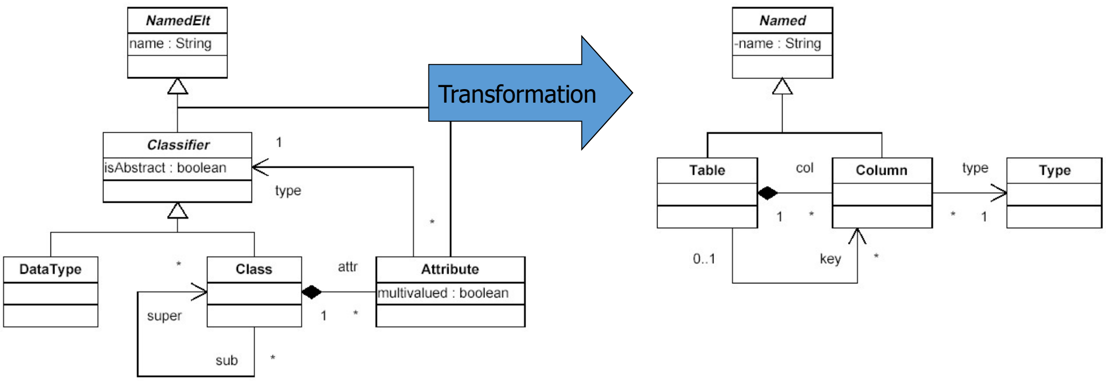
- A template that prints the class name, its comments and attributes

WebLog_fr.uml | uml2toXhtml.mt

```
<%
metamodel http://www.eclipse.org/uml2/2.0.0/UML
%>

<%script  type="uml.Class" name="uml2toXhtml" file="<%name%>.html"%>
<html>
    <head/>
    <body>
        <h1>Class Description</h1>
            <p>Name of class : <%name%></p>
            <p>Comment : <%ownedComment.body%>

        <h1>Attributes</h1>
            <%if (attribute.nSize() == 0){%>
            <p>No attributes.</p>
            <%}else{%>
            <ul>
                <%for (attribute){%>
                <li><%name%> : <%type.name%></li>
                <%}%>
            </ul>
            <%}%>
    </body>
</html>
```

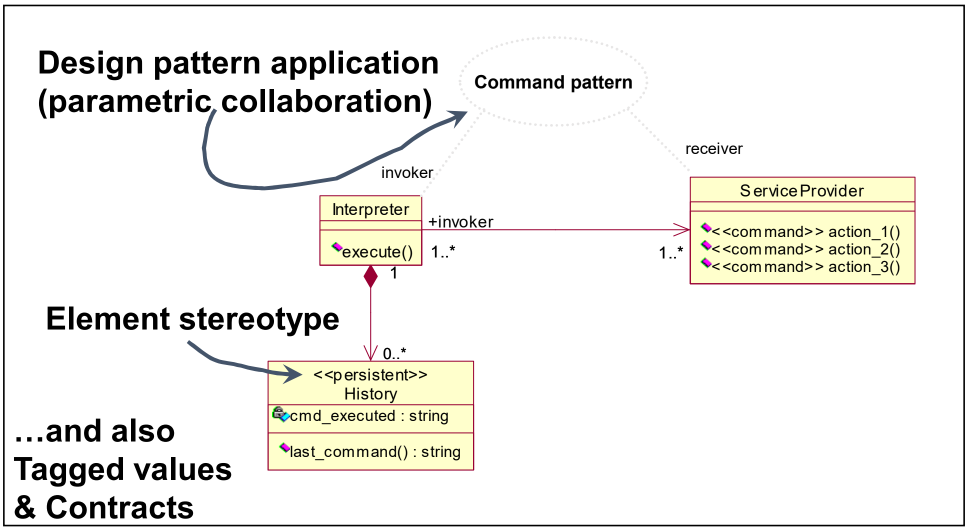

Model-to-Model: Typical Example

From UML to RDBMS



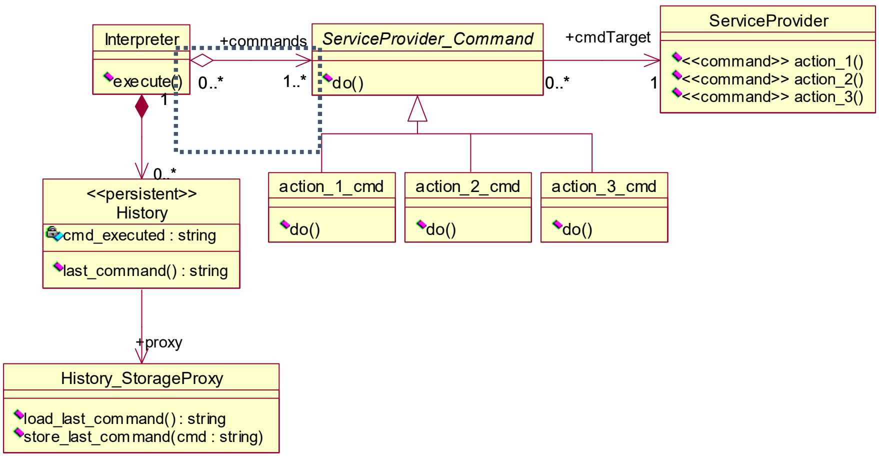
UNIVERSITÉ DE RENNES 1

M2M: Reuse Engineering Know-How (Design/Test/…)



UNIVERSITÉ DE RENNES 1

The result we want : design patterns application



UNIVERSITÉ DE RENNES 1

Classification of Model Transformation Tools

- Several approaches
  - Graph-transformation-based approaches
  - Relational approaches (aka Logic Programming)
  - Structure-Driven (OO) approaches
  - Hybrid approaches
- Rich ecosystem of tools, e.g.
  - ATL : a transformation language developed by Inria
  - GReAT : a transformation language available in the GME
  - Epsilon: for model-to-model, model-to-text, update-in-place, migration and model merging transformations.
  - Henshin: a model transformation language for EMF, based on graph transformation concepts, providing state space exploration capabilities
  - Kermeta : a general purpose modeling and programming language, also able to perform transformations
  - Mia-TL : a transformation language developed by Mia-Software
  - QVT : the OMG has defined a standard for expressing M2M transformations, called **MOF/QVT** or in short QVT.
  - SiTra: a pragmatic transformation approach based on using a standard programming language, e.g. Java, C#
  - Stratego/XT : a transformation language based on rewriting with programmable strategies
  - Tefkat : a transformation language and a model transformation engine
  - UML-RSDS [9] : a model transformation and MDD approach using UML and OCL
  - VIATRA : a framework for transformation-based verification and validation environment
  - …

UNIVERSITÉ DE RENNES 1

Model to Models in Practice

- **M2M Transformations as OO Programs**
  - **Kermeta/Xtend used to be a good choice**
  - **But now with modern Java, can be in plain Java using JMI**

Attributes

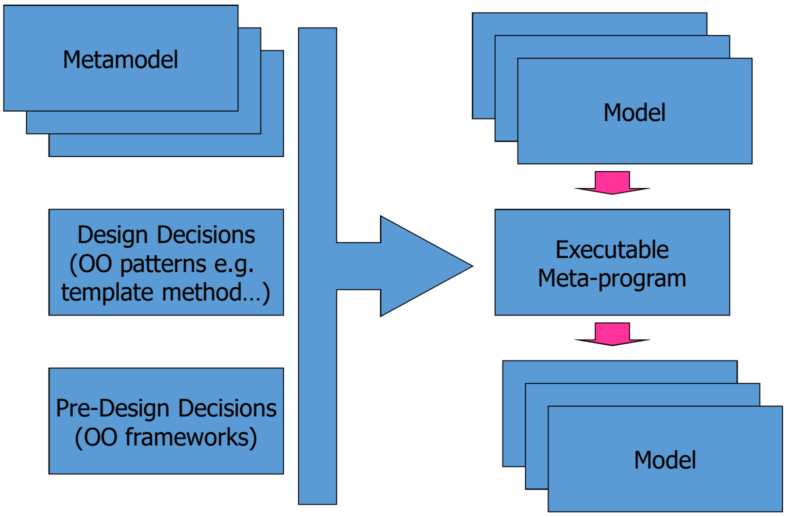
```
package javax.jmi.model;
import javax.jmi.reflect.*;
public interface Attribute extends StructuralFeature {
    public boolean isDerived();
    public void setDerived(boolean newValue);
}
```

Operations

```
package javax.jmi.model;
import javax.jmi.reflect.*;
public interface Operation extends BehavioralFeature {
    public boolean isQuery();
    public void setQuery(boolean newValue);
    public java.util.List getExceptions();
}
```

UNIVERSITÉ DE RENNES 1

General scheme

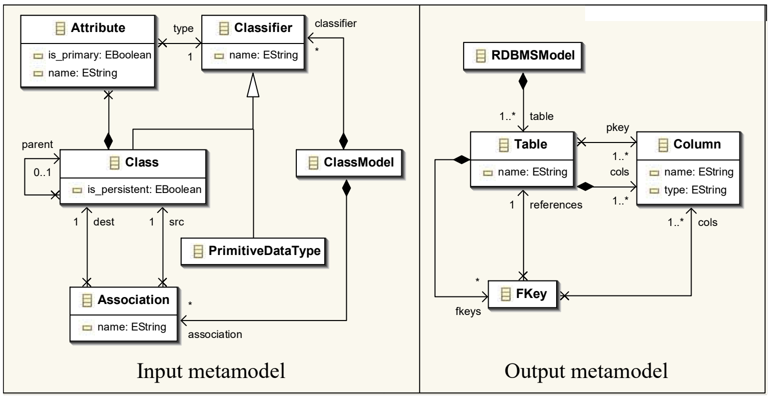


UNIVERSITÉ DE RENNES 1

“Programming style” Issues

- The transformation is simply an object-oriented program that manipulates model elements
  - Uses the OO structure of the meta-model to cleanly modularize the transformation
- OO techniques
  - Customizability through inheritance/dyn. binding
  - Pervasive use of GoF like Design Patterns

Defining the metamodels



UML2RDBMS template method

- Create tables
  - Tables are created from classes marked as persistent in the input model
- Create columns
  - For each persistent class process all attributes and outgoing associations to create corresponding columns. The foreign keys are created but the *cols* property cannot be filled and the corresponding columns cannot be created because primary keys of *references* table cannot be known before it has been processed.
- Update foreign-keys
  - The foreign-key columns are created in the table that contains the foreign-key and the property *cols* of foreign-keys is updated.

=> Handle details/variability into subclasses

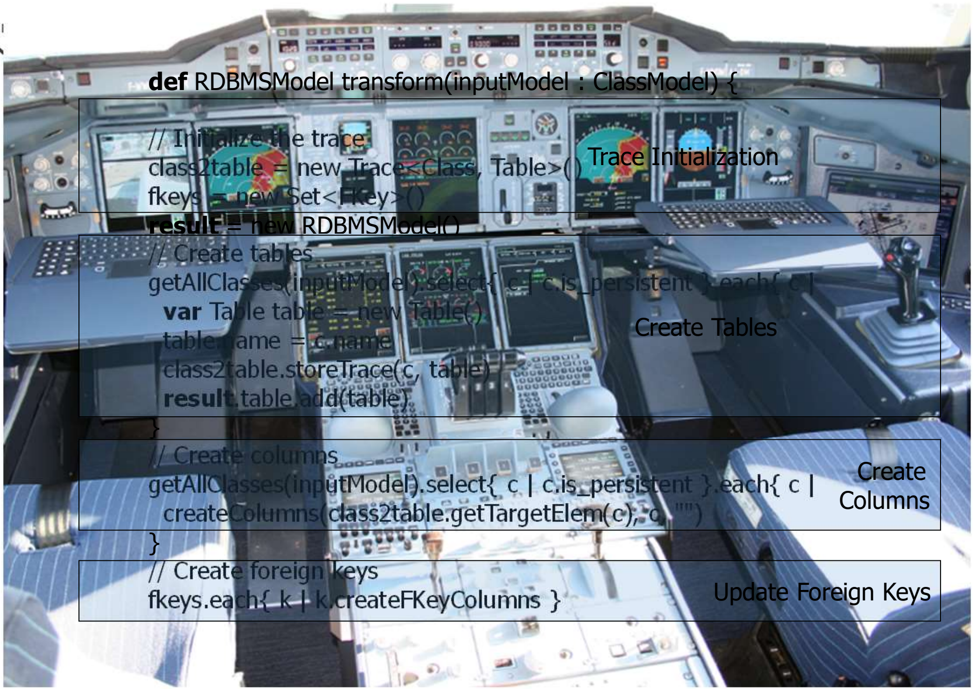
UNIVERSITÉ DE RENNES 1

Writing the transformation

Loading ECore and Kermeta metamodels

```
package Class2RDBMS;
require kermeta        // The kermerta standard library
require "trace.kmt     // The trace framework
require "../metamodels/ClassMM.ecore"  // Input metamodel in ecore
require "../metamodels/RDBMSMM.kmt" // Output metamodel in kermeta
[...]
class Class2RDBMS
{
  /** The trace of the transformation */
  reference class2table : Trace<Class, Table>

  /** Set of keys of the output model */
  reference fkeys : Collection<FKey>
[...]
```


UNIVERSITÉ DE RENNES 1



Object-orientation

- Classes and relations, multiple inheritance, late binding, static typing, class genericity, exception, typed function objects
- OO techniques such as patterns, may be applied to model transformations
  - Template method as above
  - Command, undo-redo
    - Refactorings example

```
abstract class RefactoringCommand
{
    operation check() : Boolean is abstract
    operation transform() : Void is abstract
    operation revert() : Void is abstract
}
```


Software Engineering Concerns

- Modularity in the small and the large
  - classes & packages
- Reliability
  - static typing, typed function objects and exception handling
- Extensibility and reuse
  - inheritance, late binding and genericity
- V & V
  - test cases

UNIVERSITÉ DE RENNES 1

Logo to NXC Compiler

- Step 1 – Model-to-Model transformation
- 

- Step 2 – Code generation with template

UNIVERSITÉ DE RENNES 1

Step 1: Model-to-Model

- Goal: prepare a LOGO model so that code generation is a simple traversal
  - => *Model-to-Model transformation*
- Example: local2global
  - In the LOGO meta-model, functions can be declared anywhere, including (deeply) nested, without any impact on the operational semantics
  - for NXC code generation, all functions must be declared in a “flat” way at the beginning of the outermost block.
  - => implement this model transformation as a *local-to-global* aspect woven into the LOGO MM

Step 1: Model-to-Model example

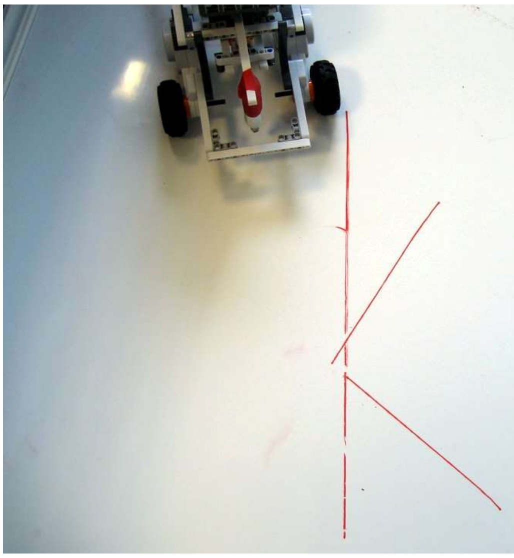
```
//  aspect local-to-global
aspect class Statement {
 def void local2global(rootBlock: Block) {
  }
}
aspect class ProcDeclaration
 def void local2global(rootBlock: Block) {
      …
  }
}
aspect class Block
   def void local2global(rootBlock: Block) {
      …
  }
}
…
```


Step 2: Model to text

- NXC Code generation using a template
  - Left as an exercise

Execution

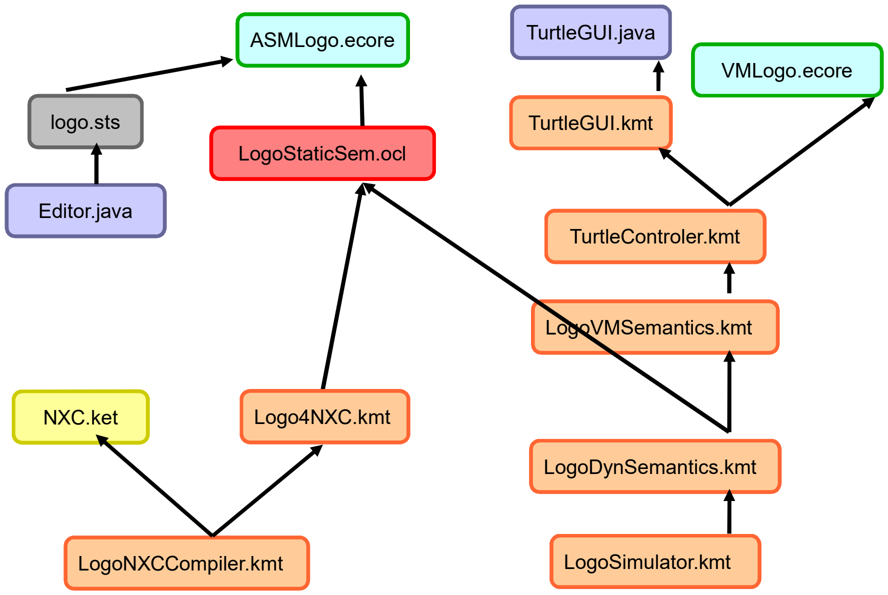
```
TO k :scale
    PENDOWN
    FORWARD *(30, :scale)
    PENUP
    BACK *(10, :scale)
    RIGHT 45
    FORWARD *(14, :scale)
    PENDOWN
    BACK *(14, :scale)
    PENUP
    RIGHT 90
    FORWARD *(14, :scale)
    PENDOWN
    BACK *(14, :scale)
    PENUP
    RIGHT 45
    FORWARD *(20, :scale)
    LEFT 180
END

CLEAR
$k(4)
```



Outline

- Introduction to Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- Operational Semantics with Kermeta
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

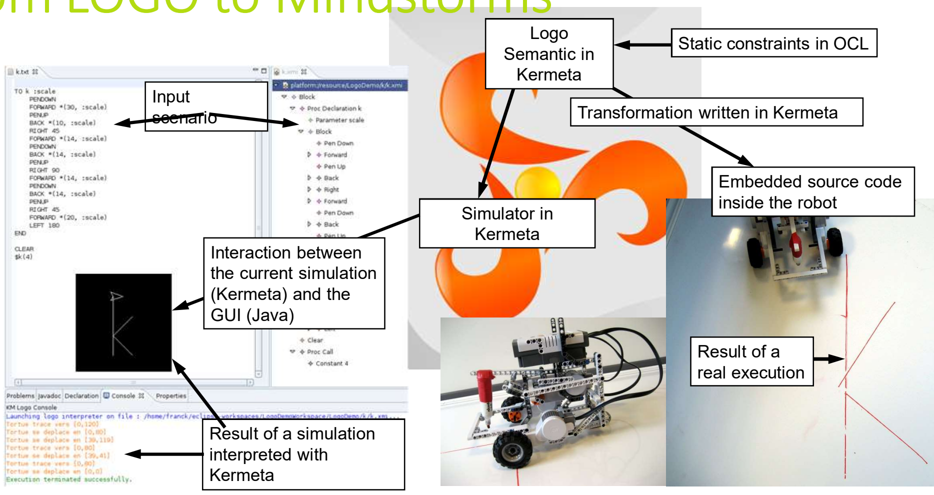
Logo Summary (1)



Logo Summary (2)

- Integrate all aspects coherently
  - syntax / semantics / tools
- Use appropriate languages
  - MOF for abstract syntax
  - OCL for static semantics
  - Kermeta for dynamic semantics
  - Java for simulation GUI
  - …
- Keep separation between concerns
  - For maintainability and evolutions

From LOGO to Mindstorms



To learn more…