UMR IRISA

Model-Driven Engineering

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

Jean-Marc Jézéquel

e-mail : jezequel@irisa.fr

http://people.irisa.fr/Jean-Marc.Jezequel

Twitter @jmjezequel

Inria — INVENTEURS DU MONDE NUMÉRIQUE

UNIVERSITÉ DE RENNES 1 — ueb

Triskell Metamodeling Kernel www.kermeta.org — Kermeta

Job description

Apple is looking for a software engineer for the modeling team focussing on autonomous technologies. The team builds model-driven development and code generation tools targeting system analysis, planning and integration.

Description

Apple is looking for software developers to help

- Design domain-specific languages that match the requirements of the individual teams,
- Implement algorithms for model analysis and planning,
- Implement code/configuration generators for the different use cases,
- Suppo

Education Details

BS or MS in Computer Science, Computer Engineering or a significant experience with language engineering.

Key Qualifications

To succeed within this role, you should have solid experience in several of the following areas:

- Software engineering and object-oriented programming (e.g. Java, C++, Swift)
- Model-driven development and code generation (e.g. Domain-specific tools, Matlab/Simulink, Labview)
- Domain-specific Language (DSL) Engineering, UML, SysML
- DSL Frameworks - e.g. Eclipse EMF, Jetbrains MPS, etc.
- Systems engineering and architectures in the context of networked, embedded systems
- Excellent Communication skills - oral, written, presentations

JobID: 113432758

Seniority Level
Not Applicable

Industry
Consumer Electronics

Employment Type
Full-time

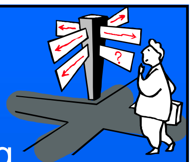
Job Functions
Engineering

See less

Airbus

- **Junior Model Based Systems Development Team Member**
  - » As part of the Model Based Systems Engineering Development Team "MBSD", you will support fulfilling the ADS global engineering model based development vision by participating to the extended organization, and contribute motivating project teams to achieve a high level of performance and quality in delivering model based development projects that provide exceptional business value to users. You will contribute to several concurrent high visibility development projects using advanced modeling methods in a fast-paced environment that may cross multiple business lines.
- **Required skills**
  - » Undergraduate or graduate degree in a technical field, and first experience in MBSE field.
  - » First experience with meta-modeling and model transformation between domains and/or other state of the art techniques.
  - » First experience with systems engineering tools and representations (e.g., NoMagic, SysML, UML, or similar).
    - – …

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

Who (does not) know this?

- Rovio's Angry Bird
  - more than 1.7 billion downloads
  - hundreds of millions of monthly active users
  - Revenue > $500M

How would you build Angry Birds?

- Only from a technical perspective
  - Leaving away the Art Design & brilliant marketing
- The game is physics-based
  - you adjust the trajectory and power of the slingshot with your finger
- Architecture?

Additional issues

- Frameworks: Box2d, PlayN
- Plateform: Android, Chrome, webOS, iOS, Mac, Maemo, Symbian, PlayStation Portable, PlayStation 3,Windows, Windows Phone, Bada
- Versions: Angry Birds, Angry Birds Seasons, Angry Birds Rio, Angry Birds Space, Angry Birds Heikki, Angry Birds Star Wars, Bad Piggies
- Pb: sync accross devices?

Philips (Medical Systems)

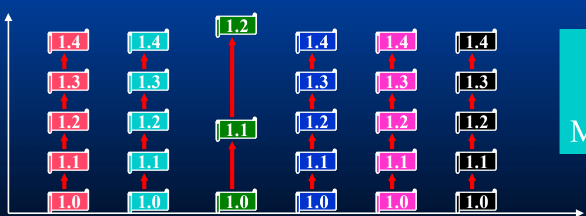
- Not just for games

Airbus

Modern Software Problems

- Importance of non-functional properties
  - distributed systems, parallel & asynchronous
  - quality of service : reliability, latency, performance...
- Flexibility of functional aspects: Product Lines
  - notion of *product lines* (space, time)

Versions (Time)

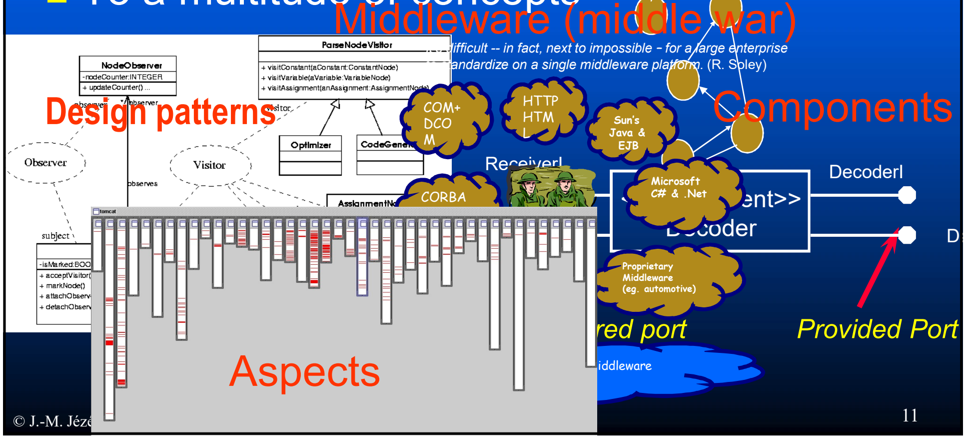


Time to Market!

Variants (Functionalities)

Once upon a time… software development looked simple

- From the object as the *only* one concept
  - As e.g. in Smalltalk
- To a multitude of concepts



Why modeling: master complexity

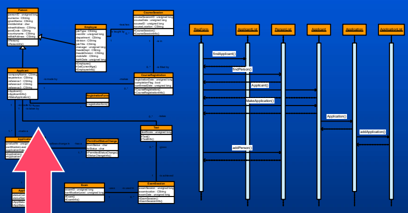
- Modeling, in the broadest sense, is the *cost-effective use of something in place of something else for some cognitive purpose*. It allows us to use something that is *simpler*, *safer* or *cheaper* than reality instead of reality for some purpose.
- A model represents reality for the given purpose;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

Specificity of Engineering: Model something not yet existing (in order to build it)



M_1 (modeling space)

Is represented by

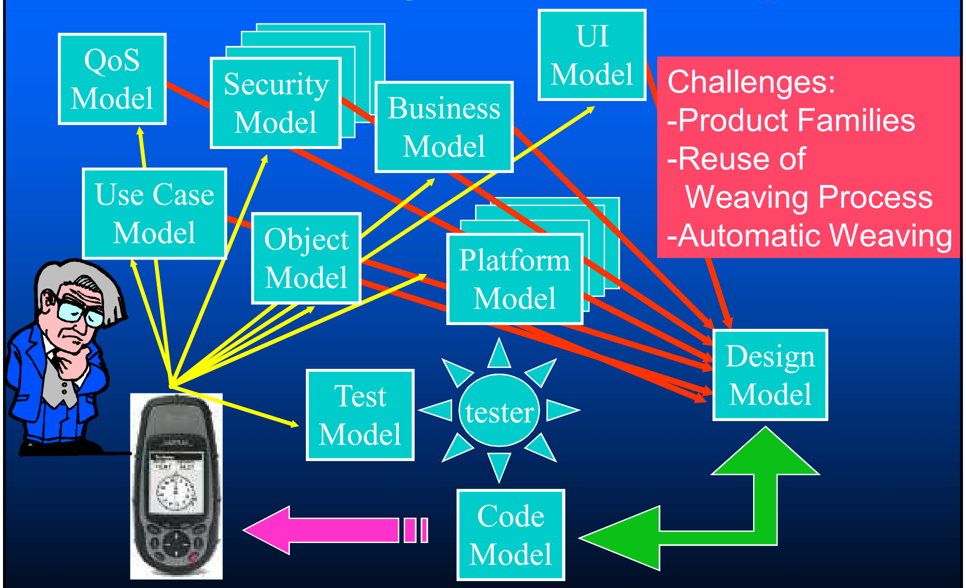
M_0 (the world)

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



© J.-M. Jézéquel, 2012-2018

15

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

© J.-M. Jézéquel, 2012-2018

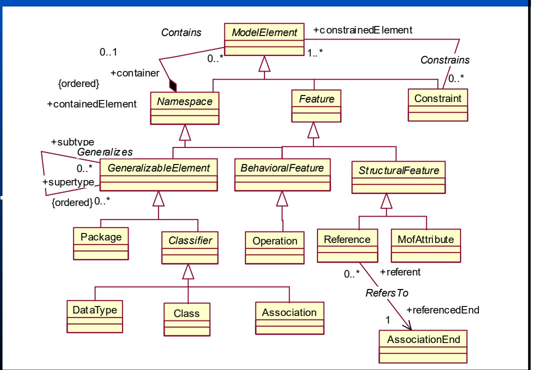
16

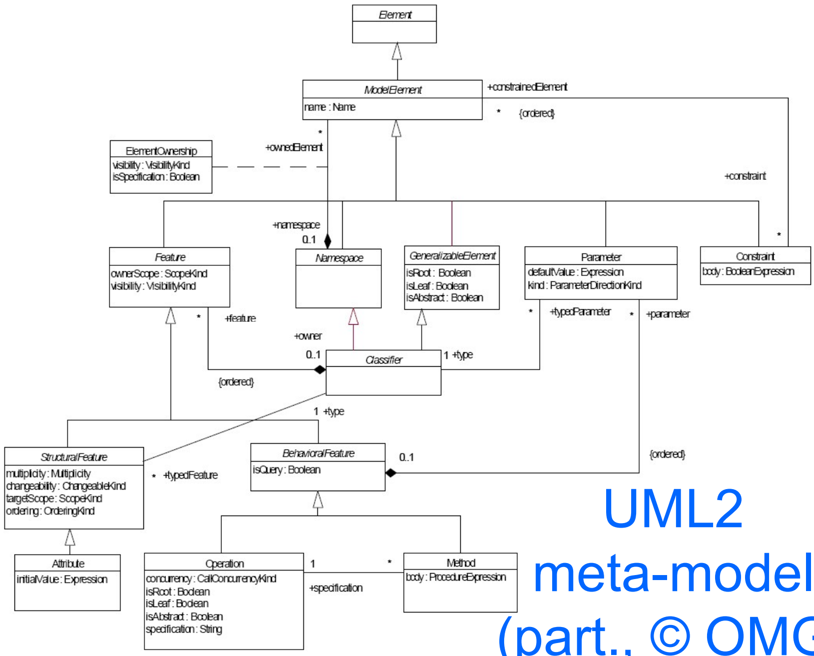
Modeling Languages

- General Purpose Modeling Languages
  - UML and its profiles (MARTE for RT…)
- Domain Specific Modeling Languages
  - Airbus, automotive industry…
  - Matlab/Simulink
- General Purpose Programming Languages
  - With restrictions (not everything allowed)
    - » GWT (Google Web Toolkit)
- Annotations, aspects…
- *In any case, Need for Language Processors*

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





UML2 meta-model (part., © OMG)

Generalizations

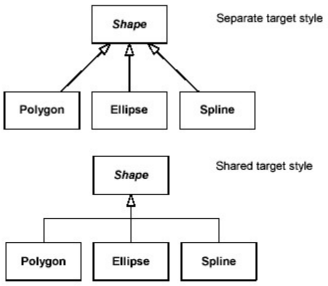


Figure 3-33. Examples of generalizations between classes.

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

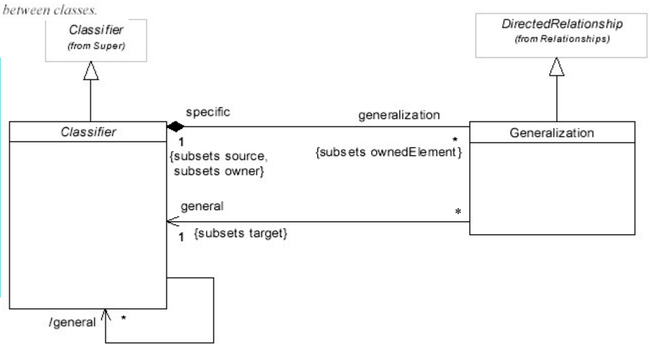
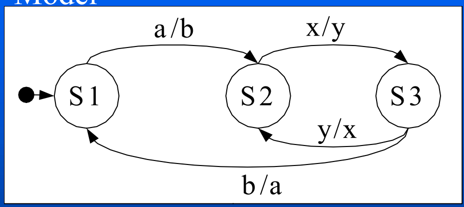


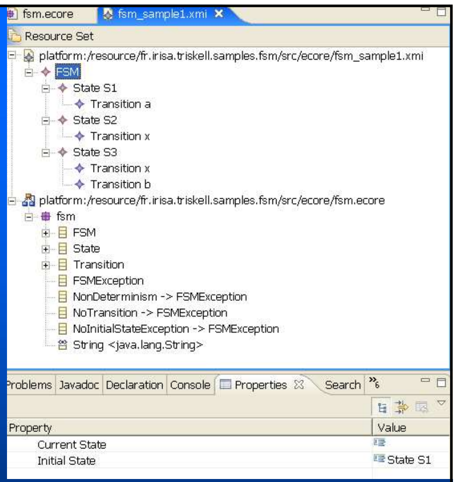
Figure 3-32. The elements defined in the Generalizations package.

© OMG

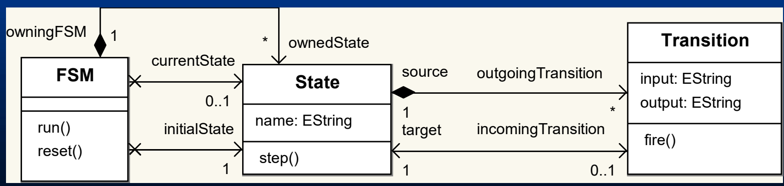
Example with StateMachines

Model

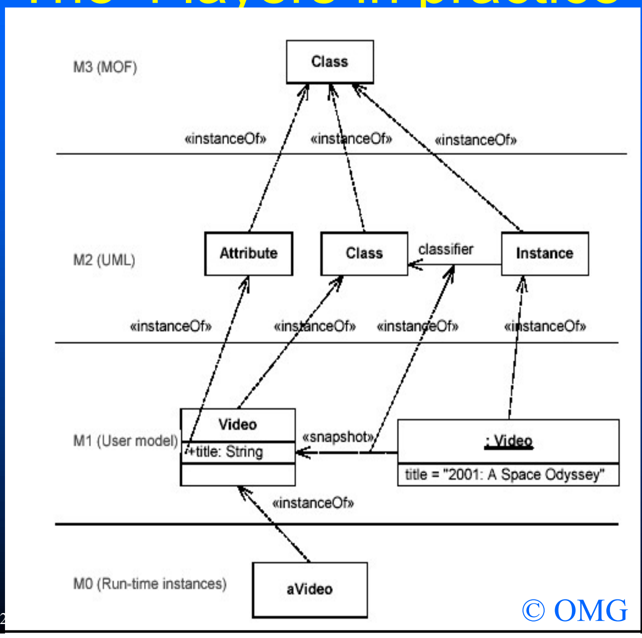




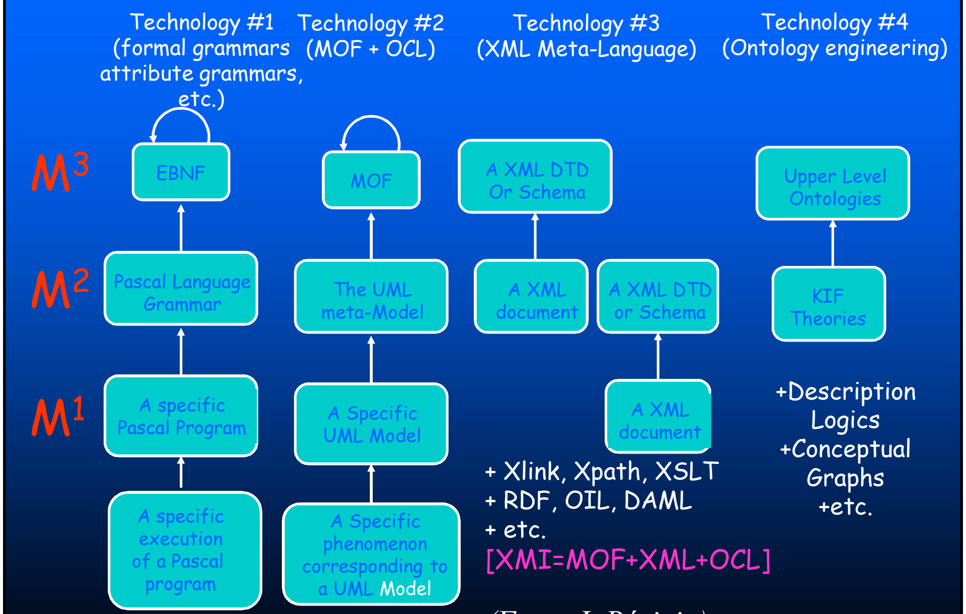
Meta-Model



The 4 layers in practice



Comparing Abstract Syntax Systems



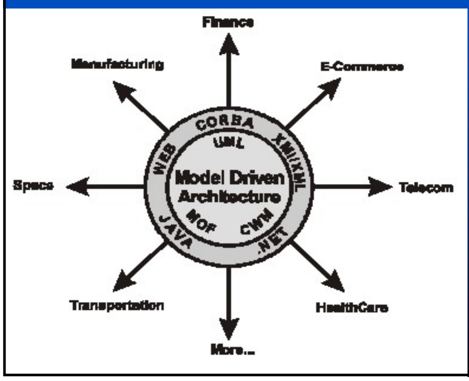
(From J. Bézivin)

© J.-M. Jézéquel, 2012-2018

23

MDA: the OMG vision

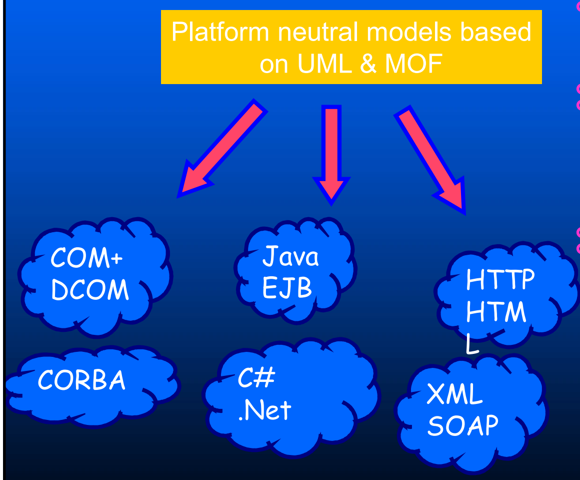
"OMG is in the ideal position to provide the model-based standards that are necessary to extend integration beyond the middleware approach… Now is the time to put this plan into effect. Now is the time for the Model Driven Architecture."



Richard Soley & OMG staff, MDA Whitepaper Draft 3.2 November 27, 2000

24

Mappings to multiple and evolving platforms



- MOF & UML as the core
- Organization assets expressed as models
- Model transformations to map to technology specific platforms

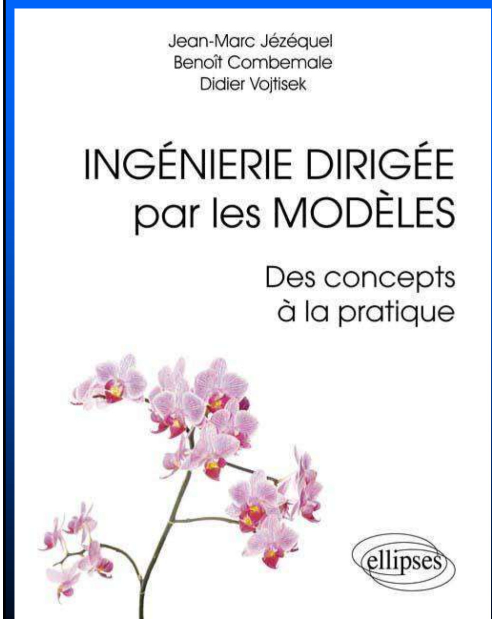
The core idea of MDA: PIMs & PSMs

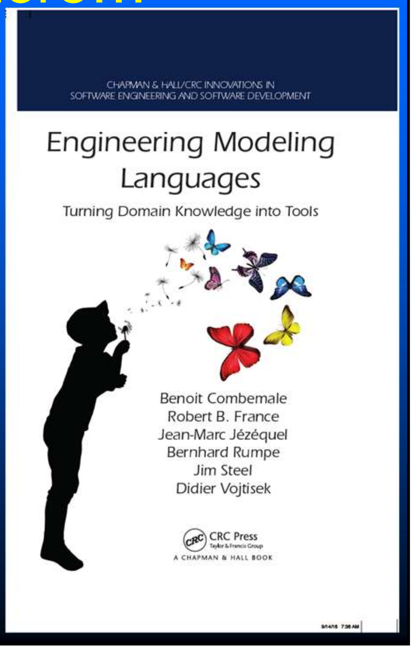
- MDA models
  - **PIM**: Platform Independent Model
    - » Business Model of a system abstracting away the deployment details of a system
    - » Example: the UML model of the GPS system
  - **PSM**: Platform Specific Model
    - » Operational model including platform specific aspects
    - » Example: the UML model of the GPS system on .NET
      - Possibly expressed with a UML profile (.NET profile for UML)
  - Not so clear about platform models
    - » Reusable model at various levels of abstraction
      - CCM, C#, EJB, EDOC, …

Model Driven Engineering : Summary

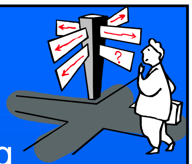
- Modeling to master complexity
  - Multi-dimensional and aspect oriented by definition
- Models: from contemplative to productive
  - Meta-modeling tools, meta-models used to define languages
- Model Driven Engineering
  - Weaving aspects into a design model
    - » E.g. Platform Specificities
- Model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

To learn more…





Outline



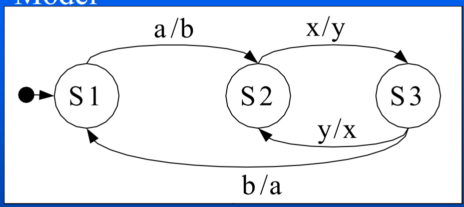
- Introduction to Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- Operational Semantics with Kermeta
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

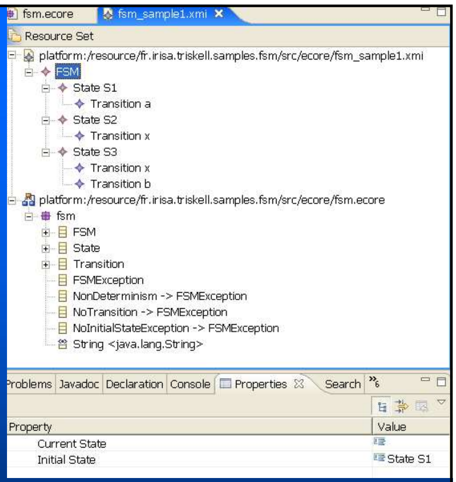
Meta-Models as Shared Knowledge

- 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…

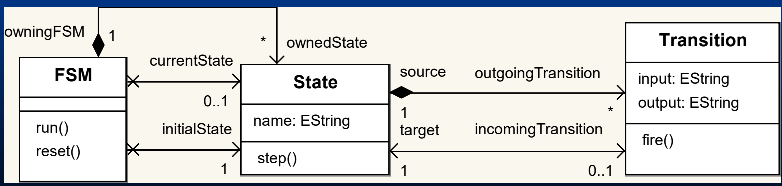
Example with StateMachines

Model

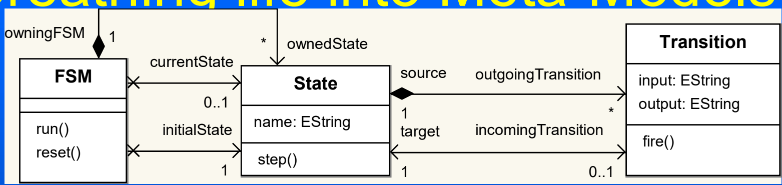




Meta-Model



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
```

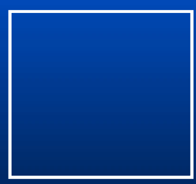
```
class Minimizer {
   public def FSM minimize (source: FSM) {…}
}
```

DIY with LOGO programs

- Consider LOGO programs of the form:

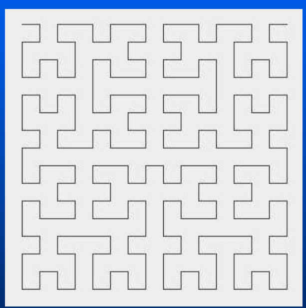
```
repeat 3  [ pendown forward 3 penup forward 4  ]
```

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



Fractals in LOGO

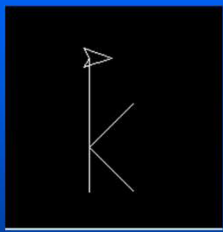
```
; lefthilbert
to lefthilbert :level :size
  if :level != 0 [
    left 90
    righthilbert :level-1 :size
    forward :size
    right 90
    lefthilbert :level-1 :size
    forward :size
    lefthilbert :level-1 :size
    right 90
    forward :size
    righthilbert :level-1 :size
    left 90
  ]
end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

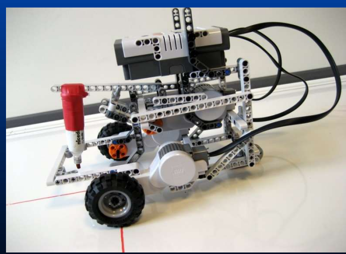


```
; righthilbert
to righthilbert :level :size
  if :level != 0 [
    right 90
    lefthilbert :level-1 :size
    forward :size
    left 90
    righthilbert level-1 :size
    forward :size
    righthilbert :level-1 :size
    left 90
    forward :size
    lefthilbert :level-1 :size
    right 90
  ]
end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

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

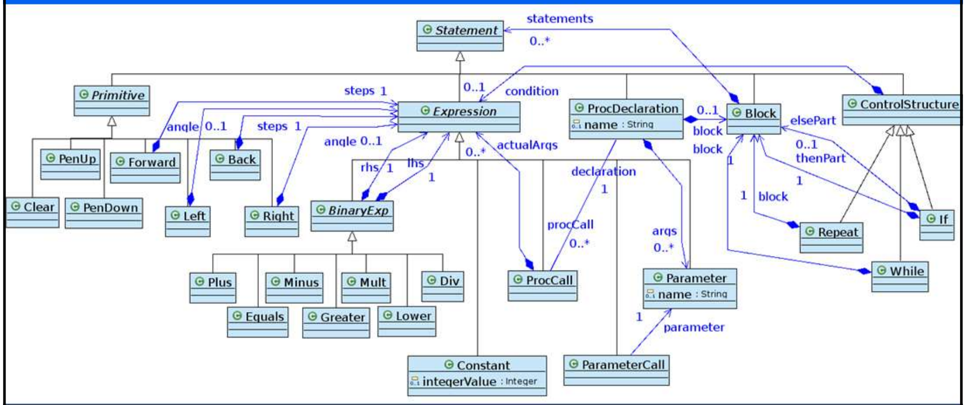
Meta-Modeling LOGO programs

- 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

ASMLogo.ecore

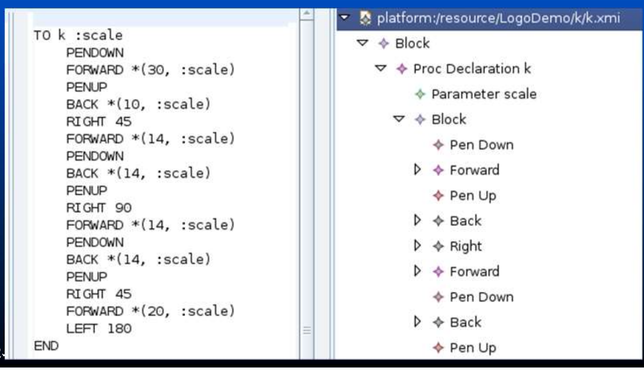
LOGO metamodel



ASMLogo.ecore

Concrete syntax

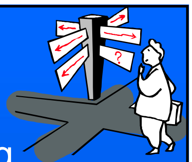
- Any regular EMF based tools
- Textual using Sintaks logo.sts
- Graphical using GMF or TopCased



Do It Yourself

- Within Eclipse
  - Load/Edit/Save Models
    - » Conforming to the LOGO meta-model
    - » ie LOGO programs
- Install & Run the MDLE4LOGO Bundle
  - On your own PC
  - Or follow the beamed demo

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)

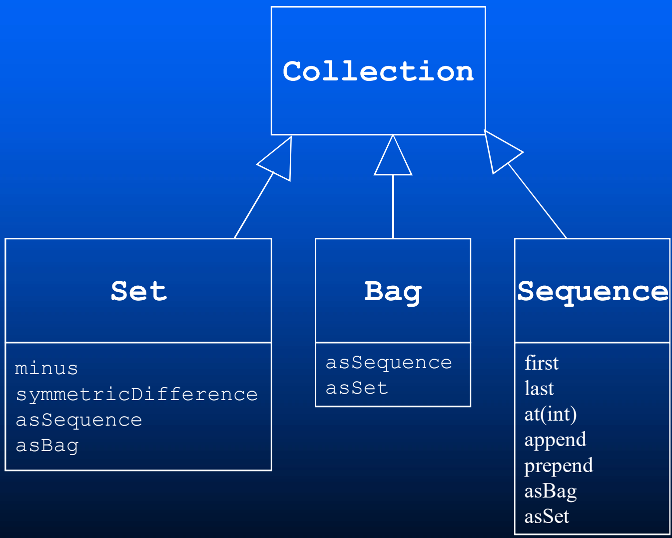
Context Car inv:
self.owner.age >= 18

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:
owings->select(v: Car | v.mileage<100000)->notEmpty
```

- shortcut:

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

- shortcut:

```
context Person inv:
owings->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:
- A procedure is called with the same number of arguments as specified in its declaration:

Static Semantics for LOGO

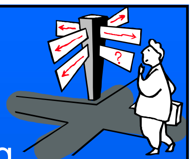
- No two formal parameters of a procedure may have the same name:

```
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
```

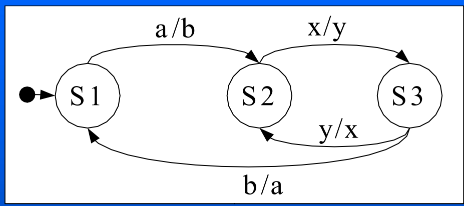

Outline



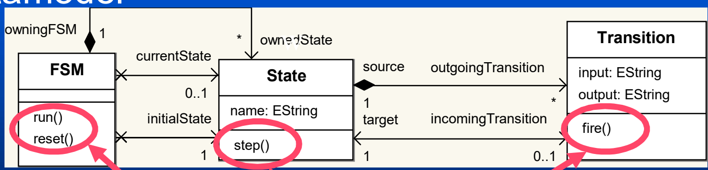
- Introduction to Model Driven Engineering
- Designing Meta-models: the LOGO example
- Static Semantics with OCL
- **Operational Semantics with Kermeta**
- Building a Compiler: Model transformations
- Conclusion and Wrap-up

Operational Semantics of State Machines

- A model



- Its metamodel

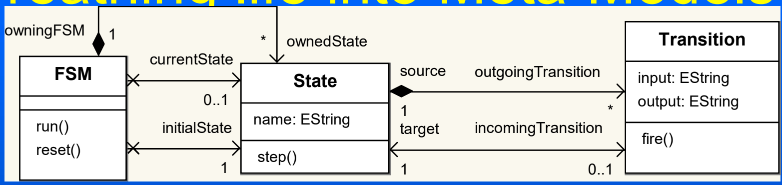


- Adding Operational Semantics to OO Metamodels

Kermeta Rationale

- Model, meta-model, meta-metamodel, DSLs…
  - Meta-bla-bla too complex for the normal engineer
- On the other hand, engineers are familiars with
  - OO programming languages (Java,C#,C++,..)
  - UML (at least class diagram)
  - May have heard of *Design-by-Contract*
- Kermeta leverages this familiarity to make Meta-modeling easy for the masses

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) {…}
}
```

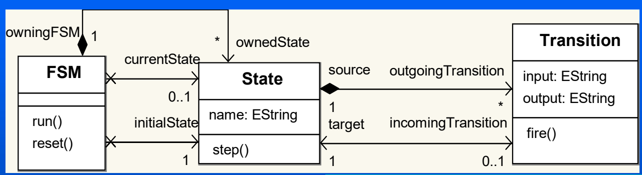

Kermeta: a **Kernel** **meta**modeling language

- Strict EMOF extension
- Statically Typed
  - Generics, Function types (for OCL-like iterators)
- Object-Oriented
  - Multiple inheritance / dynamic binding / reflection
- Model-Oriented
  - Associations / Compositions
  - Model are first class citizens, notion of model type
- Aspect-Oriented
  - Simple syntax for static introduction
  - Arbitrary complex aspect weaving as a framework
- Still “kernel” language
  - Seamless import of Java classes in Kermeta for GUI/IO etc.

Kermeta Action Language: XTEND

- Xtend = **Java 10, today!**
  - 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)

EMOF ⇔ Kermeta



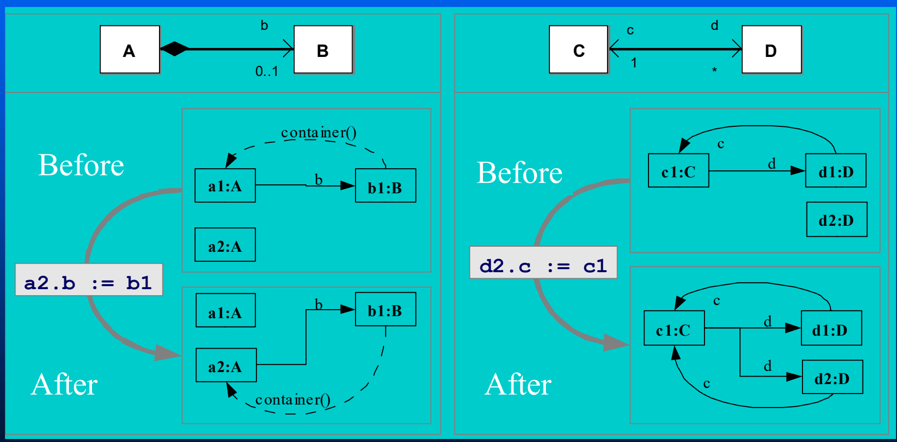

```
class FSM
{
attribute ownedState : State[0..*]#owningFSM
reference initialState : State[1..1]
reference currentState : State
operation run() : kermeta::standard::~Void is do
end
operation reset() : kermeta::standard::~Void is d
end
}
```

```
class State{
  reference owningFSM : FSM[1..1]#ownedState
  attribute name : String
  attribute outgoingTransition : Transition[0..*]#source
  reference incomingTransition : Transition#target
  operation step(c : String) : kermeta::standard::~Void is d
   end
}
class Transition{
  reference source : State[1..1]#outgoingTransition
  reference target : State[1..1]#incomingTransition
  attribute input : String
  attribute output : String
  operation fire() : String is do
  end
}
```

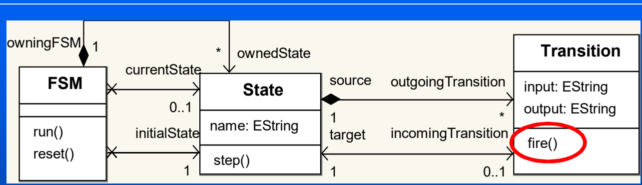
Assignment semantics

Composition

Association

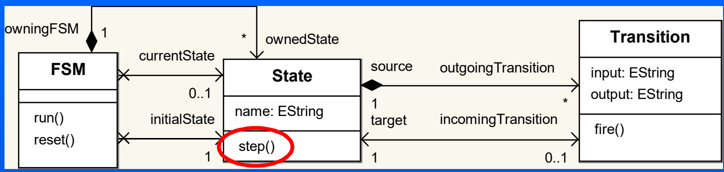


Example with Xtend

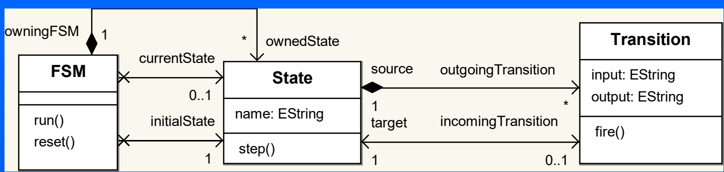


public def String fire()

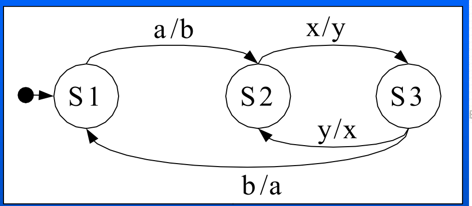
```
_self.source.owningFSM.currentState = _self.target
return _self.output
```

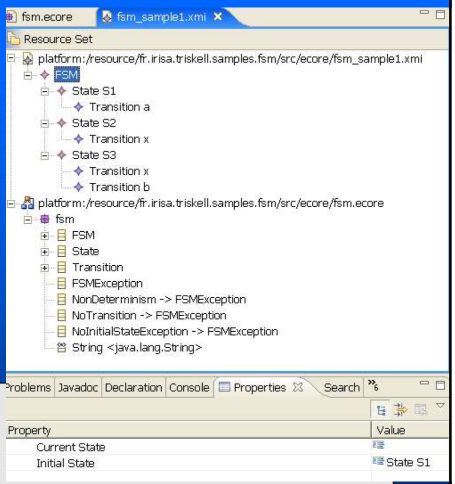



```
def String step(String c) {

 // 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"
   }
 }
```





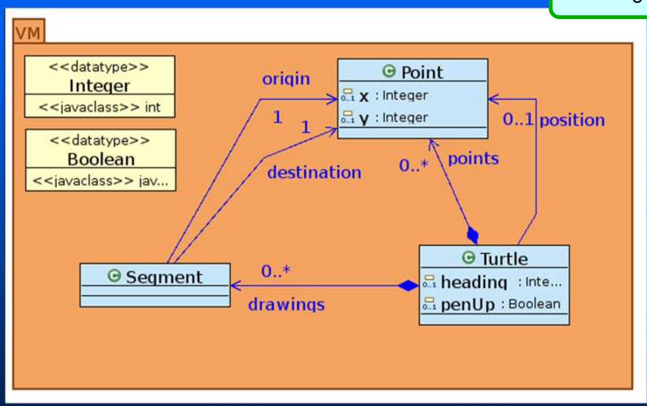

```
/**
 * Load a sample FSM from a xmi2 file
 */
operation loadFSM() : FSM is do
  var repository : EMFRepository init EMFRepository.new
  var resource : EMFResource
  resource ?= repository.createResource("../models/fsm_sample1.xmi", "../metamodels/fsm.ecore")
  resource.load

  // Load the fsm (we get the main instance)
  result ?= resource.instances.one
end
```


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

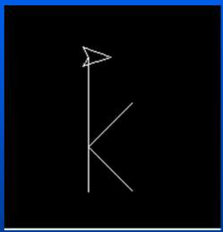
Getting an Interpreter

- 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

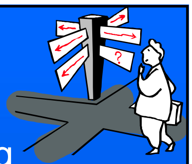
- Execute the operational semantics

```
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.
```


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

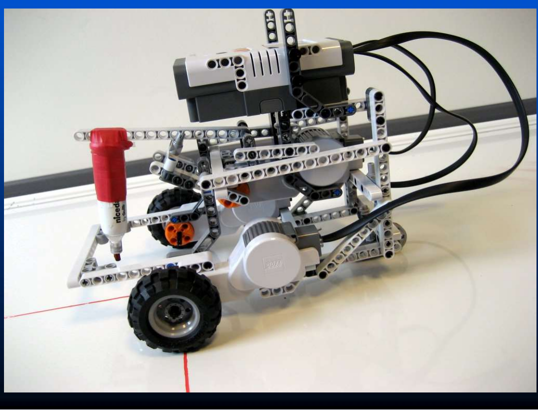
Implementing a model-driven compiler

- 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

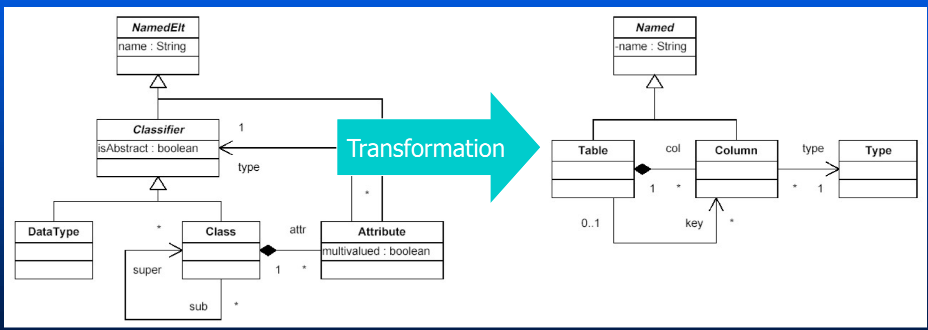
- A template that prints the class name, its comments and attributes

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>
```

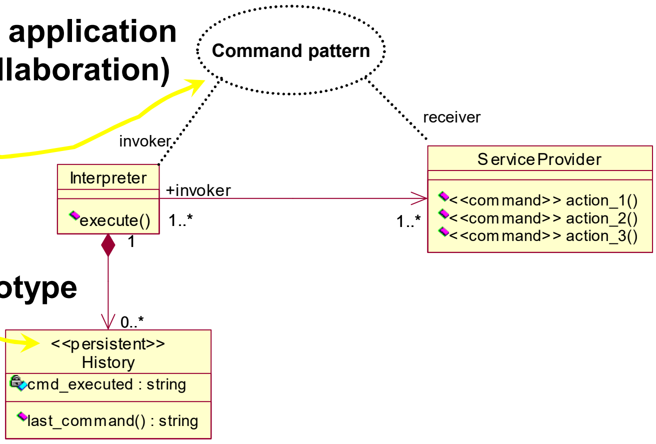
Model-to-Model: Typical Example

From UML to RDBMS



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

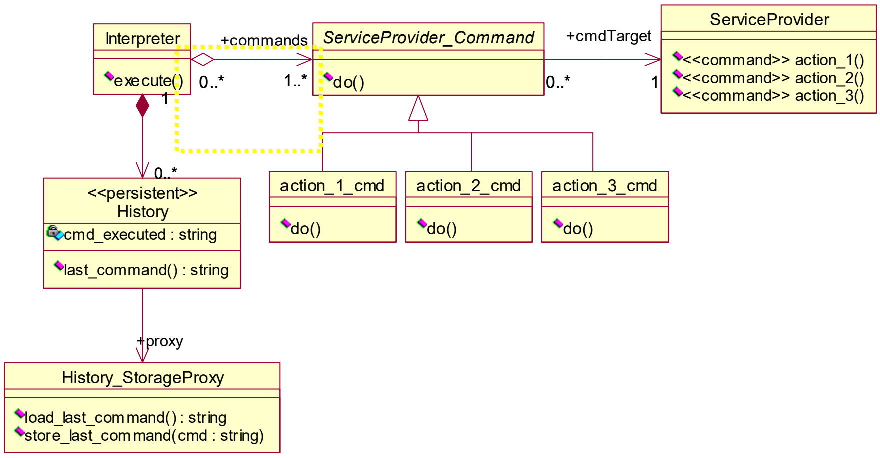
Design pattern application (parametric collaboration)



Element stereotype

…and also Tagged values & Contracts

The result we want : design patterns application



Dedicated Transformation Language

- OMG QVT style
  - Kind of DSL for transformation
- Simplify development and maintenance of model-transformations
- Higher expression power
- Enhanced structuration
  - Composition of rules
  - Interoperability

MOF 2.0 Queries/Views/Transformations RFP

- Define a language for querying MOF models
- Define a language for transformation definitions
- Allow for the creation of views of a model
- Ensure that the transformation language is declarative and expresses complete transformations
- Ensure that incremental changes to source models can be immediately propagated to the target models
- Express all new languages as MOF models

Query

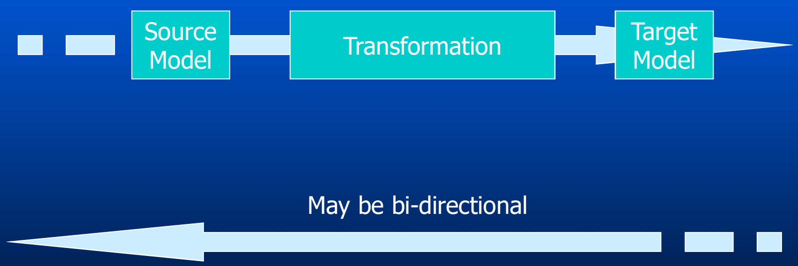
- An expression evaluated over a model
  - Returns one or more instances of types defined either in the source model or by the query language
- OCL is an example of a query language

View

- A view is a model that is completely derived from another model
  – The meta-model of the view is typically not the same as the meta-model of the source

Transformation

- A transformation generates target models from source models



Q vs V vs T

- A query is a restricted kind of view
- A view is a restricted kind of transformation
  - The target model cannot be modified independently of the source model
- A transformation generates target models from source models

Classification

- Several approaches
  - Graph-transformation-based Approaches
  - Relational Approaches
  - Structure-Driven Approaches
  - Hybrid Approaches
- Commercial
  - Mia-Transformation (Mia-Software), PathMATE (Pathfinder Solutions)
- Many academic tools
  - ATL & MTL (INRIA), AndroMDA, BOTL (Bidirectional Object oriented Transformation Language),Coral (Toolkit to create/edit/transform new models/modeling languages at run-time), Mod-Transf (XML and ruled based transformation language), QVTEclipse (preliminary implementation of some ideas of QVT in Eclipse) ou encore UMT-QVT (UML Model Transformation Tool)

Declarative

- Declarative languages describe relationships between variables in terms of functions or inference rules and the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result

Imperative

- Any programming language that specifies explicit manipulation of the state of the computer system, not to be confused with a procedural language

Declarative vs. Imperative Style

- Declarative (what to do)
  – Invariant relations between source and target models
- Imperative (how to do it)
  – How to derive a target from a source
- May be combined via pre- and post-conditions

Execution Strategy

- Invocation of the transformation rules
  – Explicit, via invocation operations (Java like)
  – Implicit, based on context and rules' signature (Prolog like)

Trace

- Trace associates one (or more) target element with the source elements that lead to its creation
  - For Round-trip development
  - Incremental propagation
- Rules may be able to match elements based on the trace without knowing the rules that created the trace

Rule

- Rules are the units in which transformations are defined
  - A rule is responsible for transforming a particular selection of the source model to the corresponding target model elements.

Declaration

- A declaration is a specification of a relation between elements in the LHS and RHS models

Implementation

- An implementation is an imperative specification of how to create target model elements from source model elements
  - An implementation explicitly constructs elements in the target model
  - Implementations are typically directed

Match

- A match occurs during the application of a transformation when elements from the LHS and/or RHS model are identified as meeting the constraints defined by the declaration of a rule
  - A match triggers the creation (or update) of model elements in the target model

Incremental

- A transformation is incremental if individual changes in a source model can lead to execution of only those rules which match the modified elements

M2M: Relational Approaches

- Declarative, based on mathematical relations
  - Good balance between flexibility and declarative expression
- Implementable with logic programming
  - Mercury, F-Logic programming languages
  - Predicate to describe the relations
  - Unification based-matching, search and backtracking

Example of logic programming

- Excerpt of Mercury code

```
conditionaltask(Id) :-
        conditionaltask_for_outputgroup_of_activity(Id, _OutputGroup).

conditionaltask_for_outputgroup_of_activity(Id, OG) :-
        outputgroup_of_activity(OG, _Activity),
        mapId(OG^og_id, conditionaltask_for_outputgroup, Id).

outputgroup_of_activity(OutputGroup, Activity) :-
        outputgroup(OutputGroup),
        contains(Activity^a_id, OutputGroup^og_id),
        activity(Activity).
```

Dedicated model transformation tools: Conclusion

- How many developers are familiar with the prolog-like style of rules writing?
- Where is the advantage of a dedicated explicit language vs. a general purpose language?
- Hybrid Languages or transformation libraries for general purpose languages…

Model to Models in Practice

- **M2M Transformations as OO Programs**
  - **Could be in plain Java, but tedious**
  - **Use Xtend (aka Java 10) instead, with JMI**

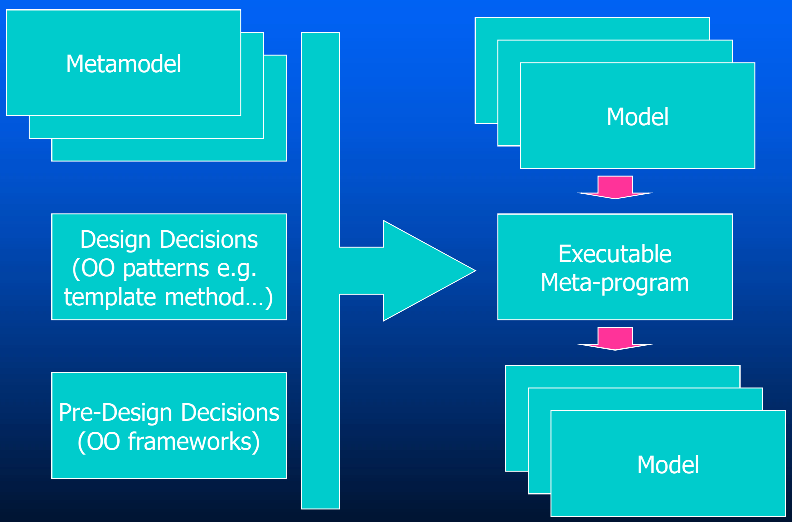
Attributes

```
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();
}
```

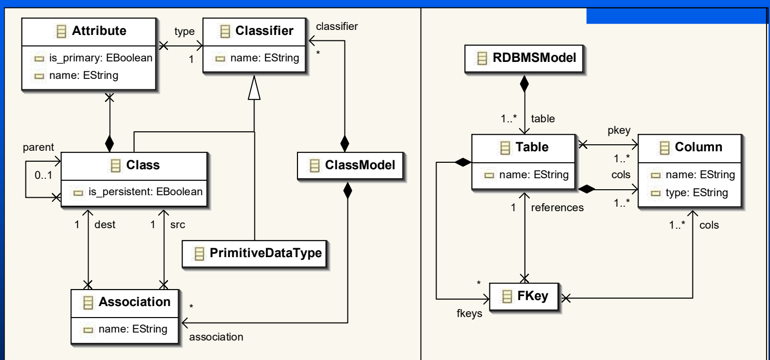
General scheme



"Programming style" Issues

- The transformation is simply an object-oriented program that manipulates model elements
  – Navigation through model is first class though (like in OCL)
- OO techniques
  – Customizability through inheritance/dyn. binding
  – Pervasive use of GoF like Design Patterns

Defining the metamodels



Input metamodel
Output metamodel

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

Writing the transformation

```
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>
[...]
```

Loading ECore and Kermeta metamodels


```
def RDBMSModel transform(inputModel : ClassModel) {

// Initialize the trace
class2table = new Trace<Class, Table>()
fkeys = new Set<FKey>()
result = new RDBMSModel()
// Create tables
getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
  var Table table = new Table()
  table.name = c.name
  class2table.storeTrace(c, table)
  result.table.add(table)
}
// Create columns
getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
  createColumns(class2table.getTargetElem(c), c, "")
}
// Create foreign keys
fkeys.each{ k | k.createFKeyColumns }
```

Trace Initialization

Create Tables

Create Columns

Update Foreign Keys

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
}
```


Composition of transformations

- Packages, classes, operations and methods, inheritance and late bindings
- Rule recursivity is handled by function recursivity

Robustness and error handling

- Kermeta is statically typed, and the code can be fully checked for correctness at compilation time.
- For unexpected behavior at runtime, the language provides exception handling.

Design variations, libraries vs. DSLs

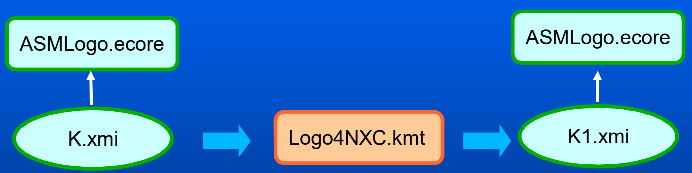
- A final design reflects a set of tradeoffs made by the developer
- The variation of the designs may be more or less constraint by the amount of pre-design and reuse provided by the language environment

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

Logo to NXC Compiler

- Step 1 – Model-to-Model transformation



- Step 2 – Code generation with template

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

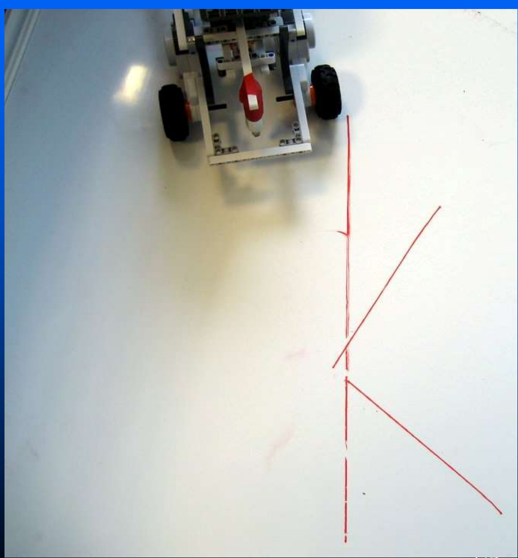
```
//  aspect local-to-global
aspect class Statement {
 def void local2global(rootBlock: Block) {
  }
}
aspect class ProcDeclaration
 def void local2global(rootBlock: Block) {
         …
  }
}
aspect class Block
   def void local2global(rootBlock: Block) {
         …
  }
}
…
```

Step 2: Model to text

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

Execution

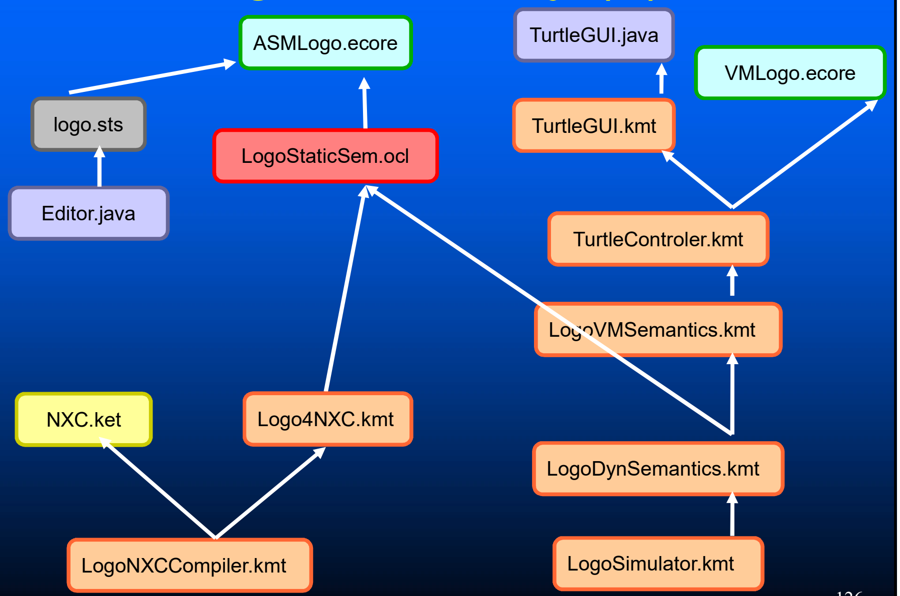
```
TO k :scale
    PENDOWN
    FORWARD *(30, :scale)
    PENUP
    BACK *(10, :scale)
    RIGHT 45
    FORWARD *(14, :scale)
    PENDOWN
    BACK *(14, :scale)
    PENUP
    RIGHT 90
    FORWARD *(14, :scale)
    PENDOWN
    BACK *(14, :scale)
    PENUP
    RIGHT 45
    FORWARD *(20, :scale)
    LEFT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

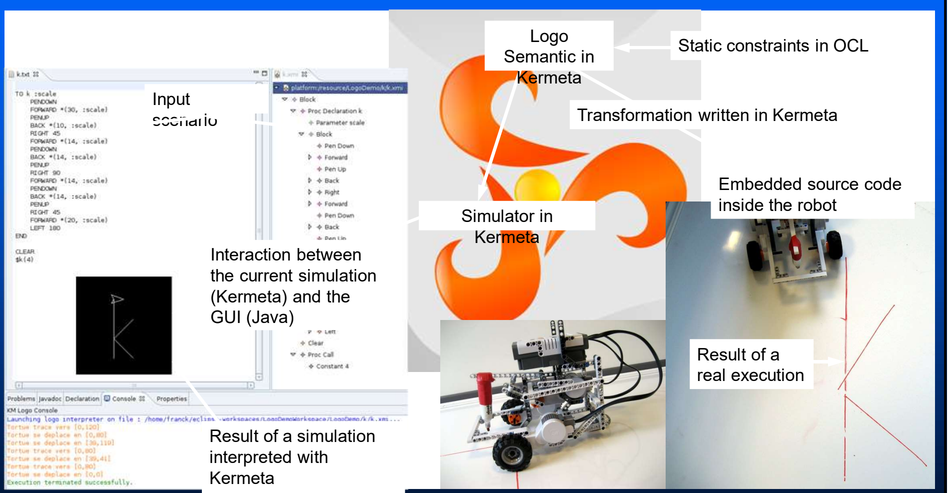
Logo Summary (1)



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



Thank you !

- Questions ?