Introduction to Model-driven Development

Martin Monperrus, Ph.D.

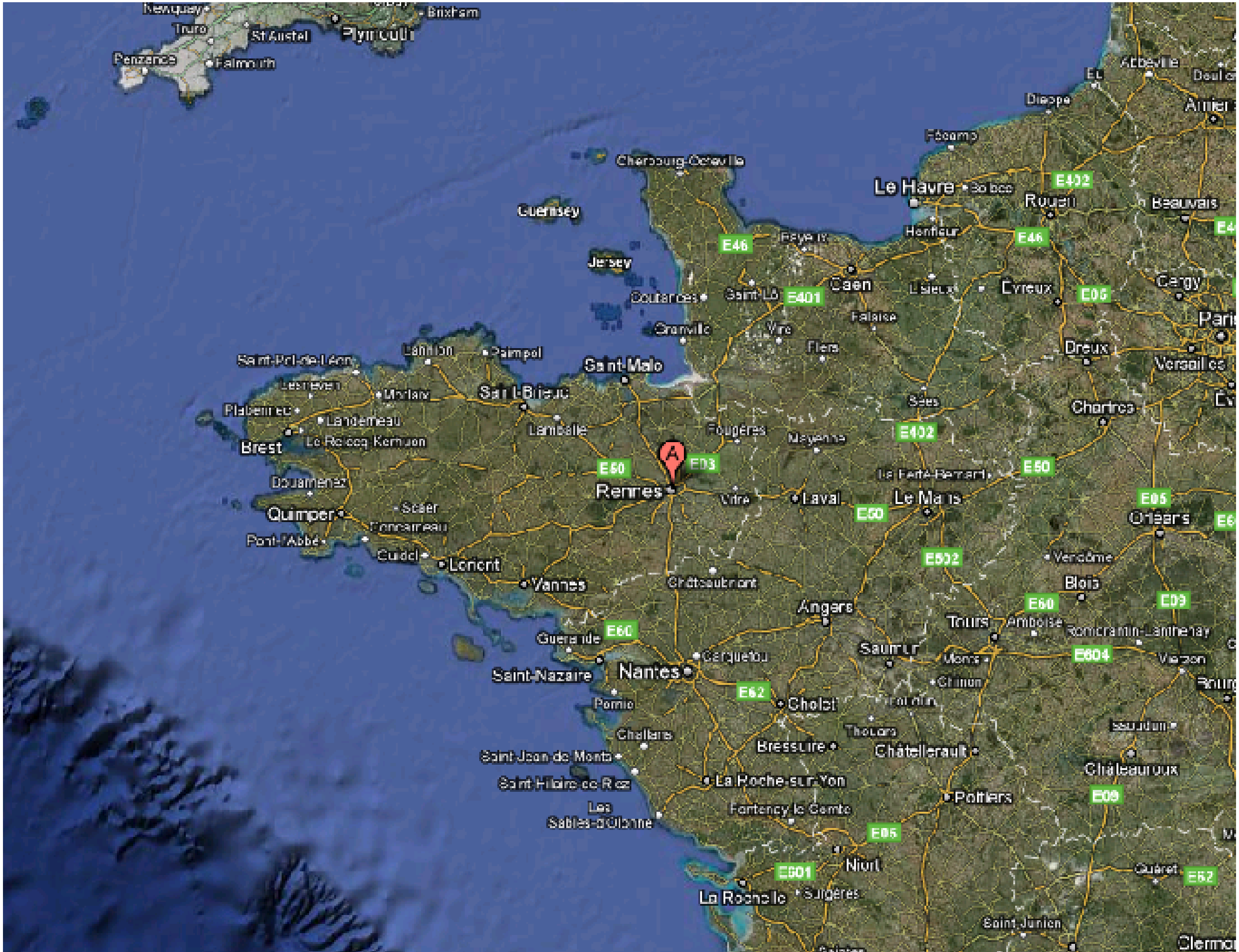
Technische Universität Darmstadt

These slides present an introduction to model-driven development.

Thanks to J-M. Jézéquel, B. Combemale and Lior Limonad for their inspiring slides.

--Martin Monperrus, June 29, 2010

Rennes, Brittany



Background

N° d'ordre: 3785

Thèse

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention INFORMATIQUE

par

Martin MONPERRUS

Équipe d'accueil : INRIA/Triskell - ENSIETA/Dtn
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

La mesure des modèles par les modèles : une approche générative

Soutenue le 6 octobre 2008 devant la commission d'examen.
Composition du jury :

Présidente			
Françoise	ANDRÉ	Professeur à l'Université de Rennes 1	
Rapporteurs			
Stéphane	DUCASSE	Directeur de Recherche à l'INRIA	
Houari	SAHRAOUI	Professeur à l'Université de Montréal	
Examinateurs			
Dominique	LUZEAUX	Directeur à la DGA	
Joël	CHAMPEAU	Enseignant-chercheur à l'ENSIETA	
Jean-Marc	JÉZÉQUEL	Professeur à l'Université de Rennes 1	(Directeur de thèse)
Invités			
Brigitte	HOELTZENER	Enseignant-chercheur à l'ENSIETA	
Gabriel	MARCHALOT	Ingénieur à Thales Airborne Systems	

Softw Syst Model
DOI 10.1007/s10270-010-0165-9

REGULAR PAPER

Model-driven generative development of measurement software

Martin Monperrus · Jean-Marc Jézéquel · Benoit Baudry · Joël Champeau · Brigitte Hoeltzener

Received: 13 October 2009 / Revised: 25 February 2010 / Accepted: 21 April 2010

Abstract Metrics offer a practical approach to evaluate properties of domain-specific models. However, it is costly to develop and maintain measurement software for each domain-specific modeling language. In this paper, we present a model-driven and generative approach to measuring models. The approach is completely domain-independent and operationalized through a prototype that synthesizes a measurement infrastructure for a domain-specific modeling language. This model-driven measurement approach is model-driven from two viewpoints: (1) it measures models of a domain-specific modeling language; (2) it uses models as unique and consistent metric specifications, with respect to a metric specification metamodel which captures all the necessary concepts for model-driven specifications of metrics. The benefit from applying the approach is evaluated by four case studies. They indicate that this approach significantly eases the measurement activities of model-driven development processes.

Communicated by Prof. Antonio Vallecillo.

M. Monperrus (✉)
Technische Unierität Darmstadt, Darmstadt, Germany
e-mail: monperrus@cs.tu-darmstadt.de

J.-M. Jézéquel
University of Rennes, Rennes, France

J.-M. Jézéquel · B. Baudry
INRIA, Rennes, France

J. Champeau · B. Hoeltzener
ENSIETA, Brest, France

1 Introduction

Metrics offer a practical approach [25,48] to evaluate non-functional properties of artifacts resulting from model-driven engineering (MDE) development processes. Ledeczi et al. [25] showed that a use of the models is design-time diagnosability analysis to determine sensor coverage, size of ambiguity groups for various fault scenarios, timeliness of diagnosis results in the onboard system, and other relevant *domain-specific metrics*. More recently, a similar point of view is expressed by Schmidt et al. [48]: in the context of enterprise distributed real-time and embedded (DRE) systems, our system execution modeling (SEM) tools help developers, systems engineers, and end users discover, *measure*, and rectify integration and performance problems early in the system's life cycle.

Contrary to general-purpose programming language measurement software, domain-specific measurement software is not big enough a niche market; that is to say there are no measurement software vendors for specific domains. Hence, companies most often have to fully support the development cost of the model measurement software. Similarly to measurement software packages for classical object-oriented programs [28], model measurement software is complex and costly. Indeed, it must address the following requirements: the automation of measurement, the integration into a modeling tool, the need for extensibility and tailoring, etc. In all, the order of magnitude of the cost of model measurement software is several man-months [28,50].

Our goal is to address the cost of measurement software for models by providing a generative approach that can synthesize a measurement environment for most kinds of models. In other words, we would like to have a prototype that allows the generation of measurement software for UML models, for AADL models, for requirements models, etc.

Published online: 13 June 2010 Springer

Model-driven software development is

- an approach to software development
- which does not rely on general purpose programming languages only
- which uses models as first-class artifacts (mostly software architecture models, domain specific models)
- which heavily uses code generation

Nota bene: for sake of simplicity, consider that
model-driven development = model-driven engineering = model-driven architecture

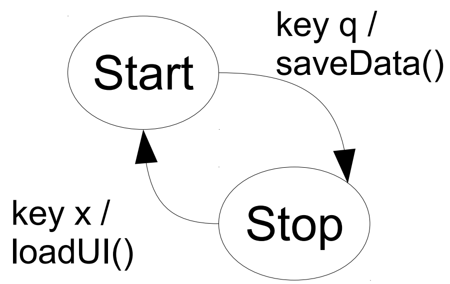
Part 1: Introduction

```
lstates[999] = (state *) malloc (sizeof (state));
current_state = lstates[999];
current_state->name = 999;
current_state->ntransitions = 25;
current_state->ltransitions =
  (transition *) malloc (sizeof (transition) * 25);
current_state = lstates[0];
t = &(current_state->ltransitions[0]);
t->trigger = 'y';
t->message = 'z';
t->tostate = lstates[543];
```

Introduction example

Problem

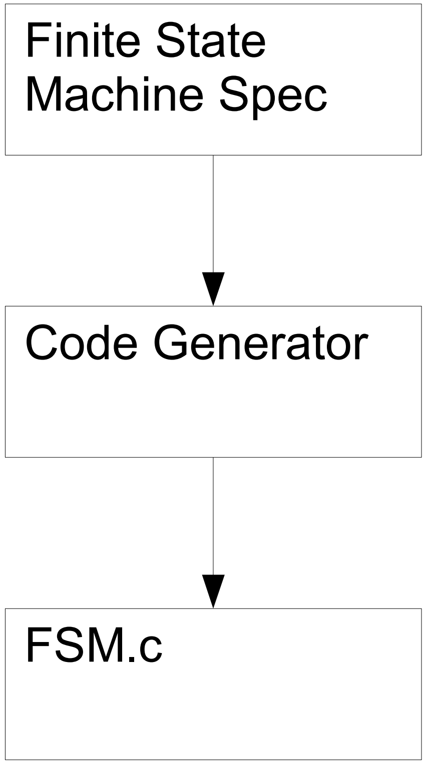
Finite State Machine



Expertise? Performance? Reuse?

C only platform

Solution



Introduction example

```
## this is python code
class Core:
  class FSM:
    def __init__(self):
      # a list of states
      self.lstates=[]
  class State:
    def __init__(self):
      # a list of transitions
      self.transitions=[]
      # a name
      self.name='unset'
  class Transition:
    def __init__(self):
      # the target state
      self.tostate = None
      # a char triggering the transition
      self.trigger=None
      # a string to be displayed when the transition is selected
      self.message=None
```

Introduction example

```
# this is a code generator
def fsm2C(fsm,cfile,printdebug):
  f = open(cfile,'w')
  for s in fsm.lstates:
    f.write('#define '+str(s.name)+' '+str(i)+"\n")
  f.write('int main(int argc, char *argv[]) {')
  f.write('do {')
  f.write('switch (state) {')
  for s in fsm.lstates:
    f.write('case '+str(s.name)+': ')
    f.write('switch (message) {')
    for t in s.transitions:
      f.write('case \''+chr(ord(t.trigger))+'\': state = '+str(t.tostate)+';\n')
      f.write('printf("-> '+t.message+'\\n");\n')
  f.write('}}\n')
  f.write('while (message !=EOF);')
  if printdebug:
    f.write('printf("Ending...\\n");')
  f.write('return 0;}\n')
```

Introduction example

```
# Definition of the model/program
# Imperative manner
fsm = Factory().createFSM()
s1 = Factory().createState()
s2 = Factory().createState()
fsm.addState(s1)
fsm.addState(s2)
t1= Factory().createTransition()
t1.trigger='x'
t1.message='yeahhh'
t1.tostate=s2
t2= Factory().createTransition()
t2.trigger='y'
t2.message='cool'
t2.tostate=s1
s1.addTransition(t1)
s2.addTransition(t2)

# ***** CODE GENERATION ****
fsm2C(fsm,'output.c',False)
```

Generated code: $indent < output.c

```
#define s668 0
#define s546 1
int
main (int argc, char *argv[])
{
//char* l=argv[1];
  long state = 0;
  char message;
  do
    {
      message = getchar ();
      switch (state)
  {
  case s668:
    switch (message)
      {
      case 'x':
        state = s546;
        printf ("-> yeahhh\n");
        break;
      default:
        break;
      }

...
```

Introduction example: declarative expression

```
# Definition of the model/program
# Imperative manner
fsm = Factory().createFSM()
s1 = Factory().createState()
s2 = Factory().createState()
fsm.addState(s1)
fsm.addState(s2)
t1= Factory().createTransition()
t1.trigger='x'
t1.message='yeahhh'
t1.tostate=s2
t2= Factory().createTransition()
t2.trigger='y'
t2.message='cool'
t2.tostate=s1
s1.addTransition(t1)
s2.addTransition(t2)
```

```
!<fsm>
lstates:
- !<state>
  name: s74
  transitions:
  - !<transition>
    trigger: x
    message: yeahhh
    tostate: s75
- !<state>
  name: s75
  transitions:
  - !<transition>
    trigger: y
    message: cool
    tostate: s74
```

The trick for marshalling

```
!<fsm>
lstates:
- !<state>
  name: s74
  transitions:
  - !<transition>
    trigger: z
    message: b
    tostate: s75
- !<state>
  name: s75
  transitions:
  - !<transition>
    trigger: l
    message: a
    tostate: s74
```

```
class MarshallingSystem:
  class FSM(Metamodel.FSM,yaml.YAMLObject):
    yaml_tag='fsm'

...
class Factory:
  def createFSM(self):
    return MarshallingSystem.FSM()

...
import yaml
stream = file('model.yaml', 'r')
fsm2 = yaml.load(stream)
stream.close()
fsm2C(fsm2,'output2.c',False)
```

Discussion

Introductory example:

- One expert in the company encodes his knowledge in a code generator (performance ↑, training ↓)
- End-users may develop software (declarative expression)
- Models can be reused (i.e. generating Java code instead of C code)

Model-driven development (MDD):

- Provides concepts to name the artifacts
- Provides tools to support the development (no use of workaround)

Source code of the example available at:
http://www.monperrus.net/martin/pyfsm-course.py

Part 2: Metamodeling

A/Prof Thomas Kühne

M.Sc TU Darmstadt, Ph.D TU Darmstadt

Position	Associate Professor
Responsibilities	Programme Director - SWEN
Research Interests	Software Engineering, Model-Driven Development, Metamodelling
Publications	Publications Listing
Office	CO233 - Postal Address
Phone	+64 4 463 5443

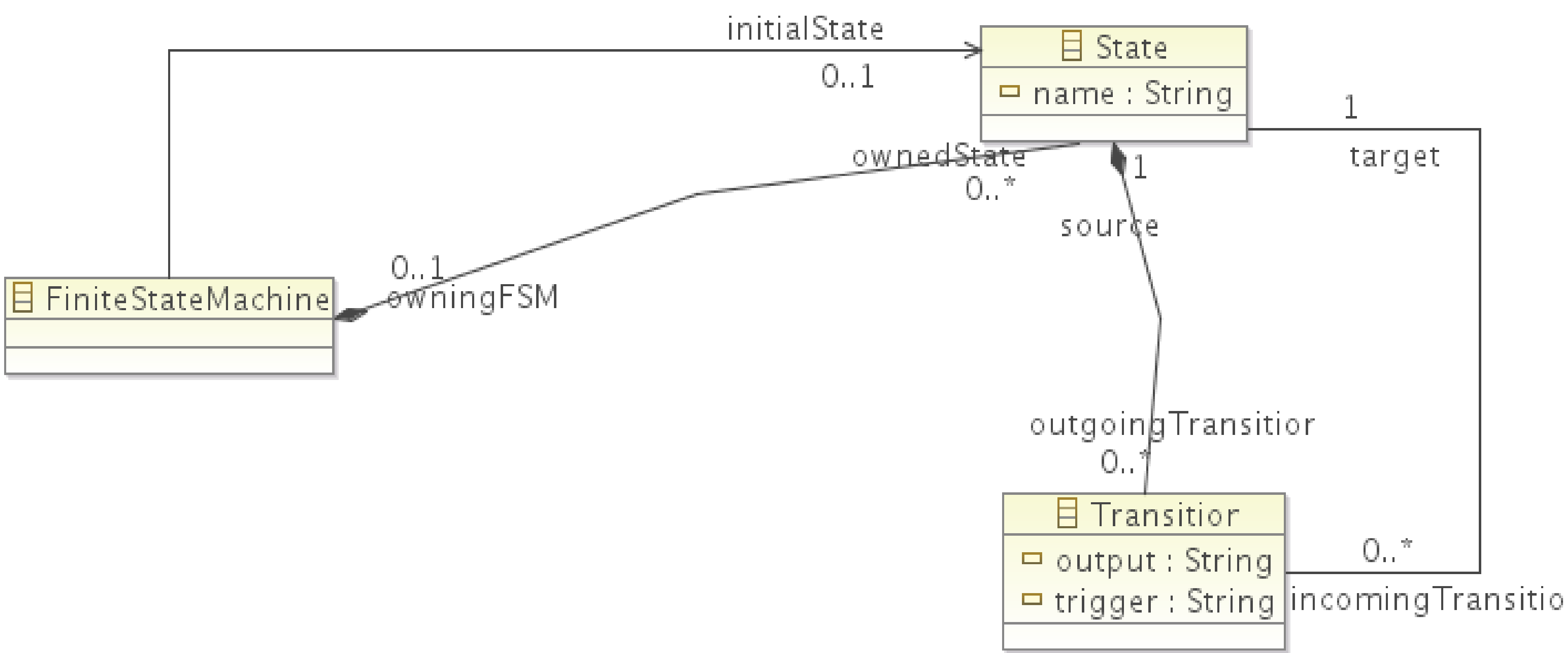
Concept: metamodel

- In MDD, a metamodel is an object-oriented model which captures the concepts and relationships of a domain.
- Instances of the metamodel are *generally* compiled to general purpose languages with code generation. They can also be interpreted.
- Instances of the metamodel are *generally* expressed declaratively.
- Instances of the metamodel are *generally* not used at runtime (no creation, no modification).

See: #Slide 7

A Metamodel for FSM

```
class Metamodel:
  class FSM:
    def __init__(self):
      # a list of states
      self.lstates=[]
  class State:
    def __init__(self):
      # a list of transitions
      self.transitions=[]
```



Metamodel for FSM

- Instances of the metamodel are *generally* compiled to general purpose languages with code generation.

```
fsm2C(fsm2,'output2.c',False)
```

- Instances of the metamodel are *generally* expressed declaratively.

```
lstates:
- !<state>
  name: s74
```

- Instances of the metamodel are *generally* not used at runtime (no creation, no modification).
  - No state or transition creation at runtime

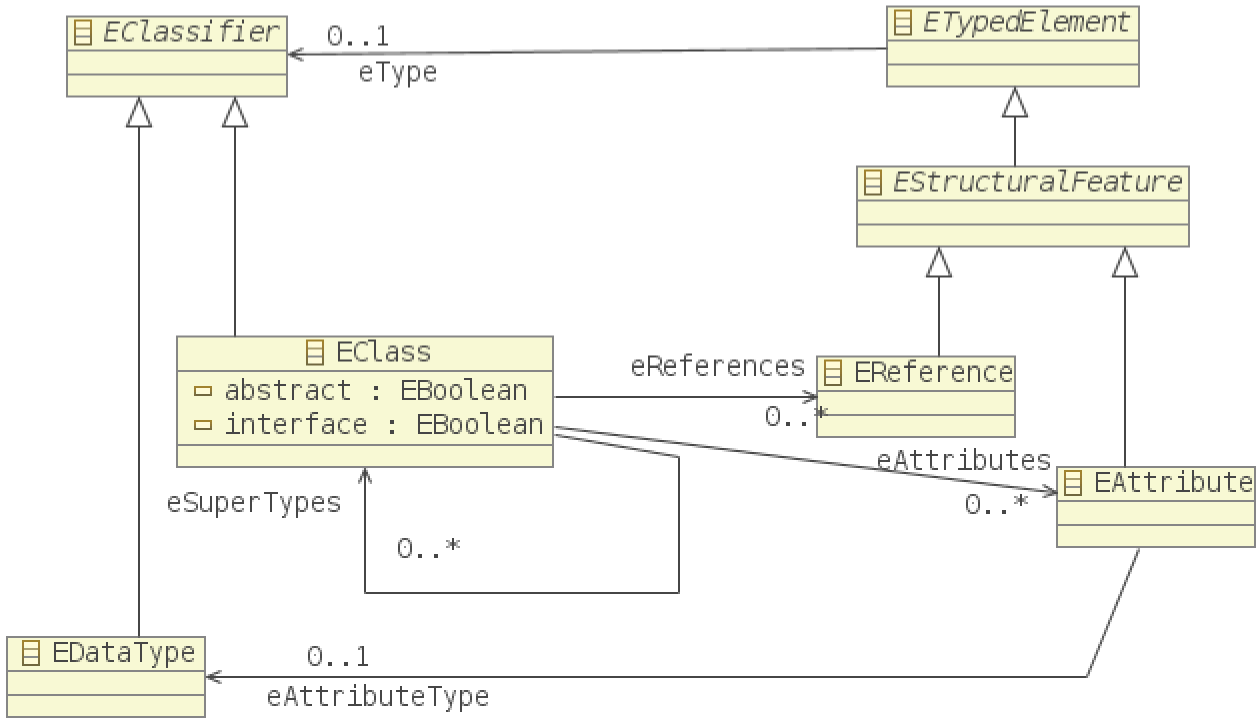
See: #Slide 7

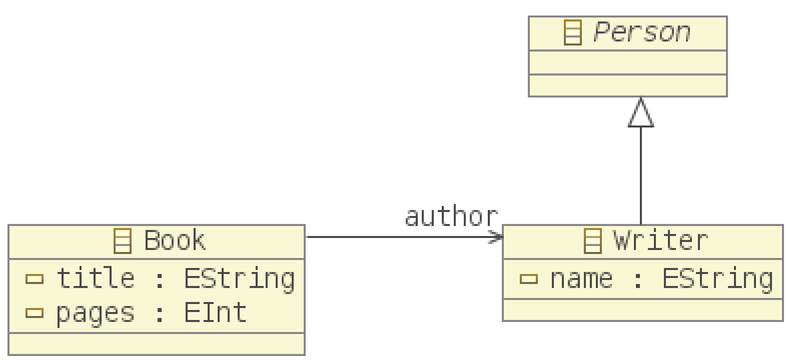
Concept: metamodel

- In MDD, a model is an instance of a metamodel.
- Models are *generally* compiled to general purpose languages with code generation. They can also be interpreted.
- Models are *generally* expressed declaratively, using a textual or graphical syntax.
- When these two points hold, a model is a kind of program.
- Models are *generally* not changed at runtime.

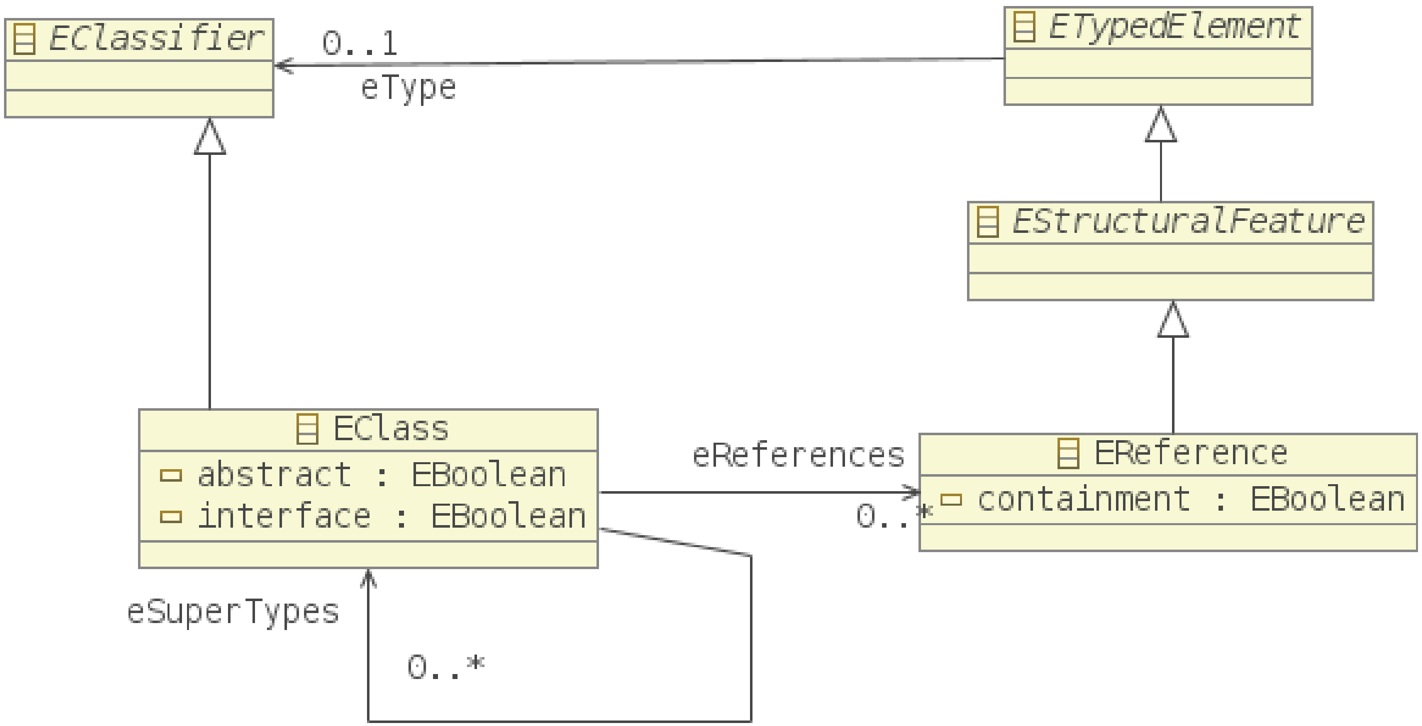
NB: In the the following, we concentrate on the Eclipse/EMF/Ecore (meta)modeling framework

Core concepts: classes, references, attributes

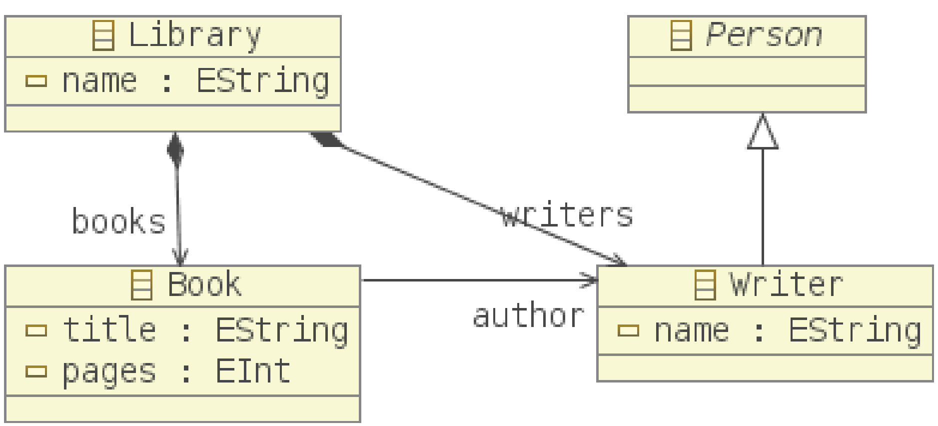




Concept: containment

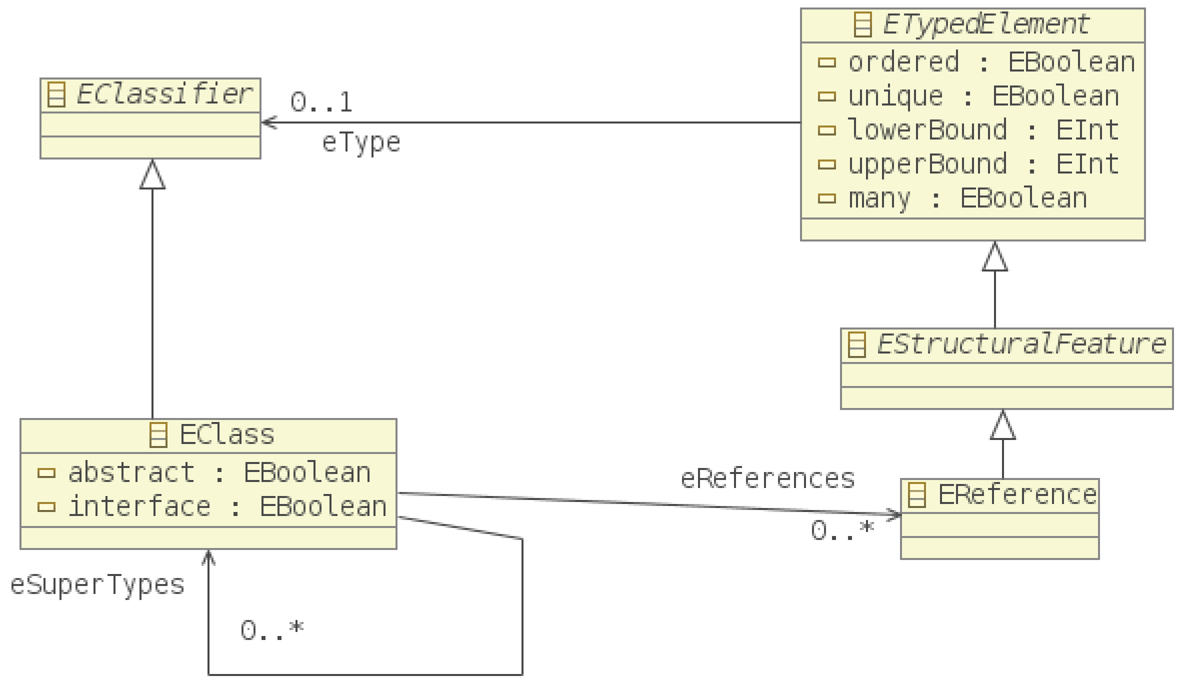


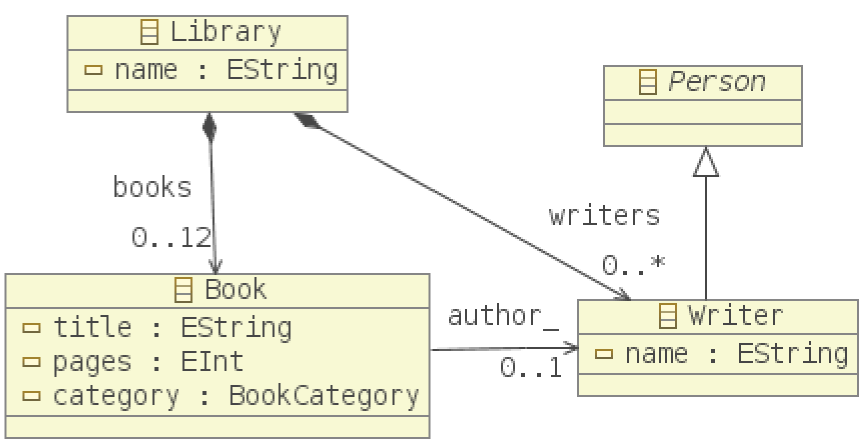
- physical meaning
- technical meaning




```xml
<library:Library>
  <writers name="Robert"/>
  <books title="Dictionnary"/>
</library:Library>
```

Concept: multiplicity (EMOF)

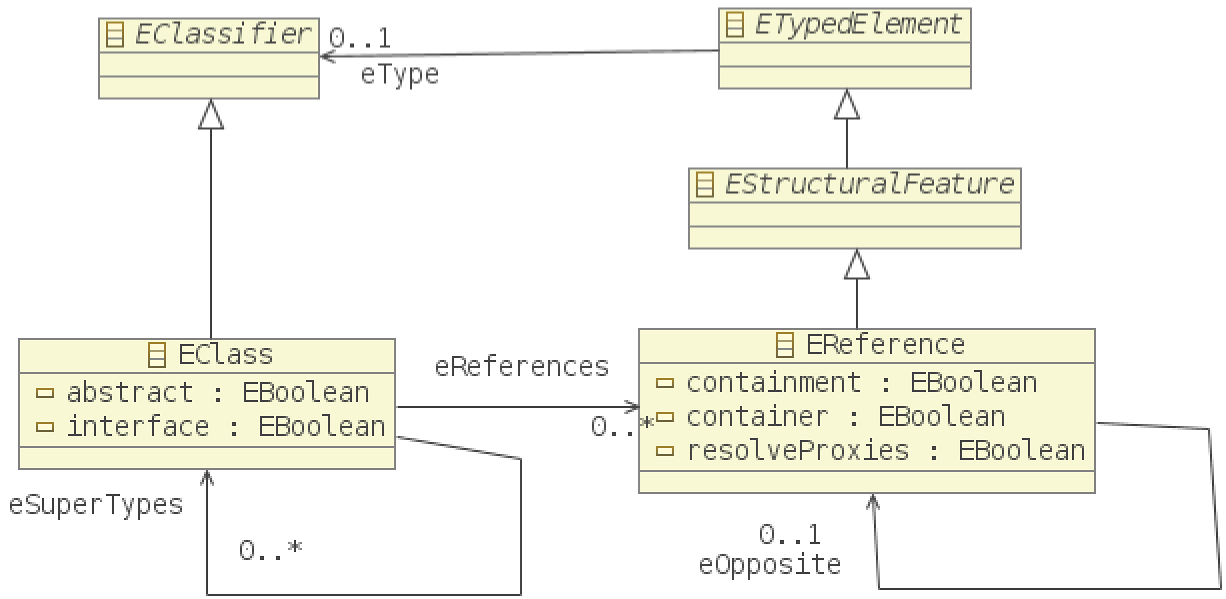




I can forget this!

ArrayList<Writer> ....
Set<Writer> ....

Concept: association/opposite/symetric references (EMOF)



I can generate this!


```
public void setAuthor(Writer newAuthor)
  {
    if (newAuthor != author)
    {

      if (author != null)
        msgs = ((InternalEObject)author).eInverseRemove(this, ..., msgs);
      if (newAuthor != null)
        msgs = ((InternalEObject)newAuthor).eInverseAdd(this, ..., msgs);
      msgs = basicSetAuthor(newAuthor, msgs);
    }  }
```

Concept: unsettable

- A feature that is declared to be unsettable has a notion of an explicit unset or no-value state. (not possible in Java)
- For example, if a boolean attribute is declared to be unsettable, it can then have any of three values: true, false, or unset.

```java
protected static final int PAGES_EDEFAULT = 100;
protected int pages = PAGES_EDEFAULT;
protected boolean pagesESet;

public int getPages() {
  if (!pagesESet) throw new RuntimeException("pages is not set");
  return pages;
}

public void setPages(int newPages) {
  pages = newPages;
  pagesESet = true;
}

public boolean isSetPages() {
  return pagesESet;
}

public void unsetPages() {
  pages = PAGES_EDEFAULT;
  pagesESet = false;
}
```

I can generate this!

Concept: model validation

- Checking whether modeled properties are satisfied (operation "validate")
  - containment, association, multiplicity
- Instead of writing consistency checking by hand

```
!<fsm>
lstates:
- !<state>
  name: s74
  transitions:
  - !<transition>
    tostate: s75
- !<state>
  name: s74
  transitions:
  - !<transition>
    trigger: l
    message: a
    tostate: s74
```

Concept: parser generation (aka marshalling)

Parsing / deserialization

```xml
<library:Library>
  <writers name="Robert"/>
  <books title="Dictionnary"/>
</library:Library>
```

```java
Resource model =
resourceSet.createResource(anotherFileURI);
model.load(null);
codeGeneration(model.getContents());
// and that's it!
```

Serialization

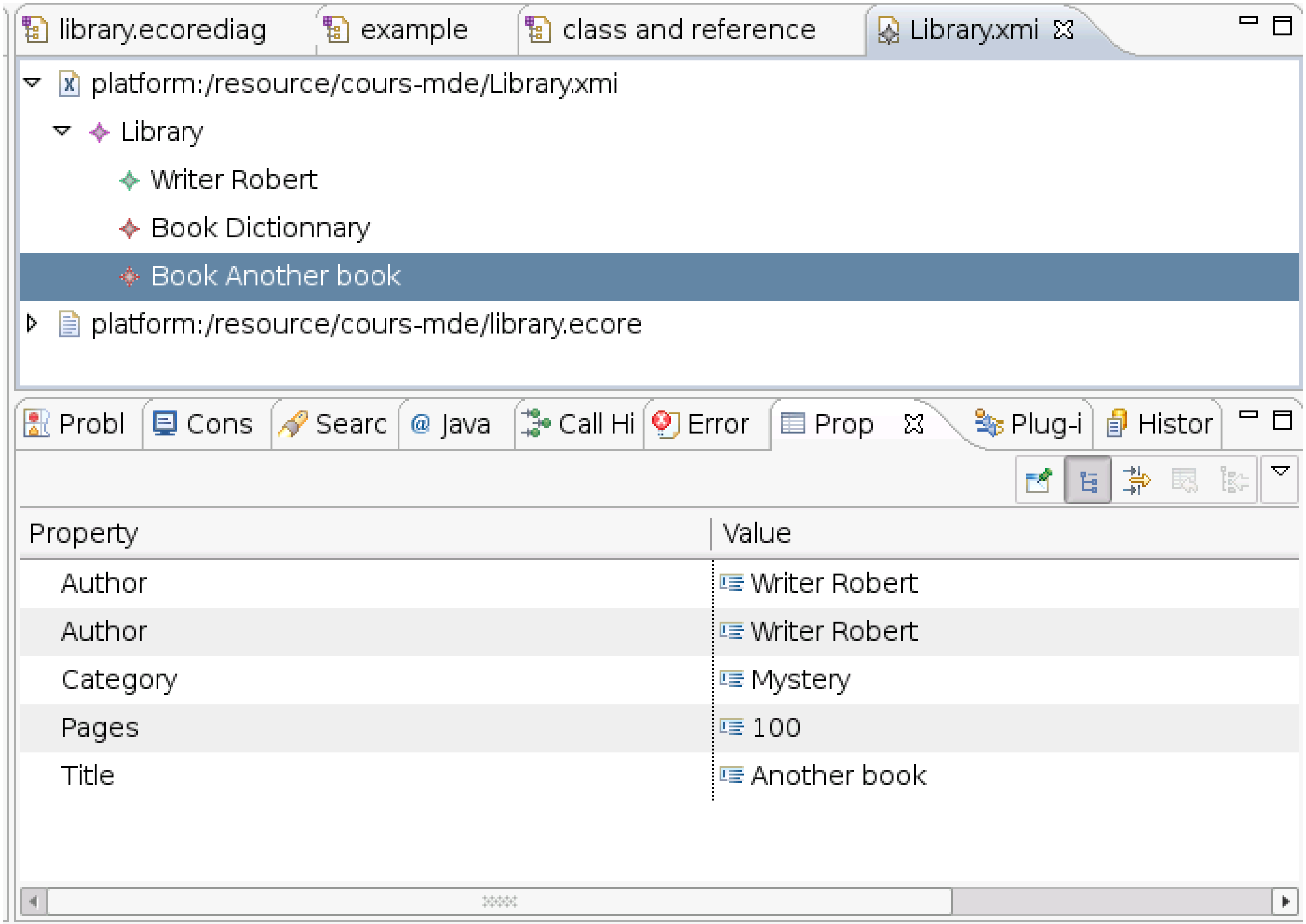
```java
Resource resource = resourceSet.createResource(anotherFileURI);
// Add the book and writer objects to the contents.
resource.getContents().add(book);
// Save the contents of the resource to the file system.
resource.save();
```

Back to the introductory example

```
!<fsm>
lstates:
- !<state>
  name: s74
  transitions:
  - !<transition>
    trigger: z
    message: b
    tostate: s75
- !<state>
  name: s75
  transitions:
  - !<transition>
    trigger: l
    message: a
    tostate: s74
```

```
!<fsm>
lstates:
- &id001 !<state>
  name: s966
  transitions:
  - !<transition>
    message: yeahhh
    tostate: &id002 !<state>
      name: s888
      transitions:
      - !<transition>
        message: cool
        tostate: *id001
        trigger: y
    trigger: x
- *id002
```

Concept: generated/semantic editor (With Eclipse/EMF)



Textual editors can be also generated if there is a grammar. See http:www.eclipse.org/Xtext/ and http://vimeo.com/8260921

Metamodeling: summary of concepts

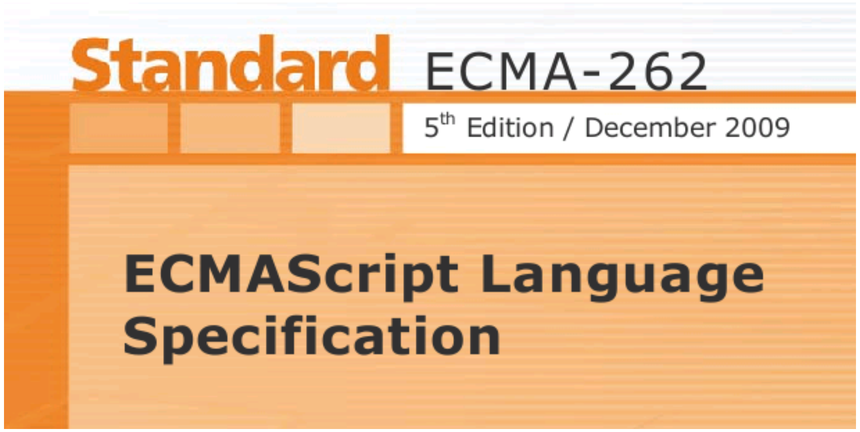
- Metamodel
- Model
- Containment for references
- Multiplicity for references
- Association / opposite references
- Unsettable datatypes and references
- Model validation
- Parser generation for declarative modeling
- Editor generation

Part 3: Execution Semantics (x5)

http://www.google.com/images?q=%22Executable+Semantics%22&start=20

Definition

- Execution semantics is the specification of the operational behavior of a model.
- The execution semantics can be formal (declarative and/or mathematics)
- The execution semantics can be executable (machine processable)



$$
\begin{aligned}
c &= \texttt{num} \mid \texttt{str} \mid \texttt{bool} \mid \texttt{undefined} \mid \texttt{null} \\
v &= c \mid \texttt{func}(x \cdots)\ \texttt{\{ return } e\ \texttt{\}} \mid \texttt{\{ } str\texttt{:}v \cdots \texttt{ \}} \\
e &= x \mid v \mid \texttt{let } (x \texttt{ = } e \cdots)\ e \mid e(e \cdots) \mid e[e] \mid e[e] \texttt{ = } e \mid \texttt{delete } e[e] \\
E &= \bullet \mid \texttt{let } (x \texttt{ = } v \cdots\ x \texttt{ = } E,\ x \texttt{ = } e \cdots)\ e \mid E(e \cdots) \mid v(v \cdots\ E,\ e \cdots) \\
&\quad \mid \texttt{\{}str\texttt{:}\ v \cdots\ str\texttt{:}E,\ str\texttt{:}e \cdots \texttt{ \}} \mid E[e] \mid v[E] \mid E[e] \texttt{ = } e \mid v[E] \texttt{ = } e \\
&\quad \mid v[v] \texttt{ = } E \mid \texttt{delete } E[e] \mid \texttt{delete } v[E]
\end{aligned}
$$

$$\texttt{let (x = v } \cdots\texttt{) e} \hookrightarrow e[x/v] \cdots \qquad \text{(E-Let)}$$

$$(\texttt{func}(x_1 \cdots x_n)\ \texttt{\{ return e \}})(v_1 \cdots v_n) \hookrightarrow e[x_1/v_1 \cdots x_n/v_n] \qquad \text{(E-App)}$$

$$\texttt{\{ } \cdots \texttt{ str: v } \cdots \texttt{ \}[str]} \hookrightarrow v \qquad \text{(E-GetField)}$$

$$\frac{str_x \notin (str_1 \cdots str_n)}{\texttt{\{ } str_1\texttt{:}\ v_1\ \cdots\ str_n\texttt{:}\ v_n \texttt{ \} } [str_x] \hookrightarrow \texttt{undefined}} \qquad \text{(E-GetField-NotFound)}$$

$$\begin{array}{c} \texttt{\{ } str_1\texttt{:}\ v_1 \cdots\ str_i\texttt{:}\ v_i\ \cdots str_n\texttt{:}\ v_n \texttt{ \} } [str_i] \texttt{ = v} \\ \hookrightarrow \texttt{\{ } str_1\texttt{:}\ v_1 \cdots\ str_i\texttt{:}\ v\ \cdots str_n\texttt{:}\ v_n \texttt{ \}} \end{array} \qquad \text{(E-UpdateField)}$$

$$\frac{str_x \notin (str_1 \cdots)}{\texttt{\{ } str_1\texttt{:}\ v_1 \cdots \texttt{\} } [str_x] \texttt{ = } v_x \hookrightarrow \texttt{\{ } str_x\texttt{:}\ v_x\texttt{,}\ str_1\texttt{:}\ v_1 \cdots \texttt{ \}}} \qquad \text{(E-CreateField)}$$

$$\begin{array}{c} \texttt{delete \{ } str_1\texttt{:}\ v_1 \cdots\ str_i\texttt{:}\ v_x\ \cdots str_x\texttt{:}\ v_n \texttt{ \} } [str_x] \\ \hookrightarrow \texttt{\{ } str_1\texttt{:}\ v_1 \cdots\ str_i\texttt{:}\ v\ \cdots str_n\texttt{:}\ v_n \texttt{ \}} \end{array} \qquad \text{(E-DeleteField)}$$

$$\frac{str_x \notin (str_1 \cdots)}{\texttt{delete \{ } str_1\texttt{:}\ v_1 \cdots \texttt{\} } [str_x] \hookrightarrow \texttt{\{ } str_1\texttt{:}\ v_1 \cdots \texttt{\}}} \qquad \text{(E-DeleteField-NotFound)}$$

Fig. 1. Functions and Objects

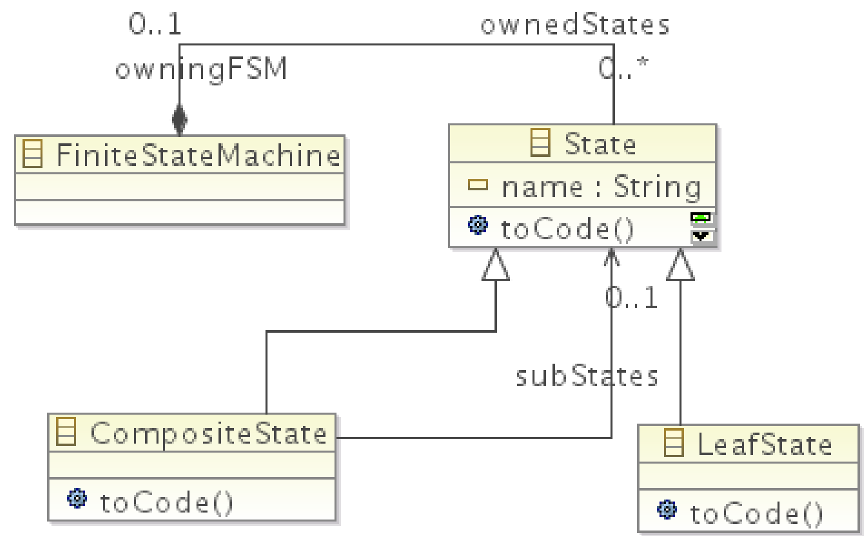
The Quick'n'Dirty Code Generation Pattern

```
# this is a code generator
def fsm2C(fsm,cfile,printdebug):
  f = open(cfile,'w')
  for s in fsm.lstates:
    f.write('#define '+str(s.name)+' '+str(i)+"\n")
  f.write('int main(int argc, char *argv[]) {')
  f.write('do {')
  f.write('switch (state) {')
  for s in fsm.lstates:
    f.write('case '+str(s.name)+': ')
    f.write('switch (message) {')
    for t in s.transitions:
      f.write('case \''+chr(ord(t.trigger))+'\': state = '+str(t.tostate)+';\n')
      f.write('printf("-> '+t.message+'\\n");\n')
  f.write('}}\n')
  f.write('while (message !=EOF);')
  if printdebug:
    f.write('printf("Ending...\\n");')
  f.write('return 0;}\n')
```

The "toString" Code Generation Pattern

- is based on methods in the metamodel and generally uses polymorphism.

```
class Intestate {
  String toCode() {
   // POLYMORPHISM
   return ownedStates.join("\n");}
}
class CompositeState {
 String toCode() {
 // the generated code uses the State pattern as
client
  return "class "+this.name+"{ State currentState
}
class LeafState {
 String toCode() {
  // the generated code uses the State pattern as
State
  return "class "+this.name+"extends State {}
}
```



The Visitor Code Generation Pattern

- Visitor pattern
  - decouples data structures and algorithms
  - generally, each class has an "accept" method
  - supports different generators

```
class FiniteStateMachine {
  void accept(IVisitor v) { v.visitFiniteStateMachine(this)}
}

class State {
  void accept(IVisitor v) { v.visitState(this)}
}

class CodeGenerator1 implements IVisitor{
  StringBuffer code;
  void visitFiniteStateMachine(FiniteStateMachine f) {
     for(State s:f.ownedStates){
       s.accept(this)}}

  void visitState(State s) {code.append(s.toString())}
}
```

```
fsm.accept(new CodeGenerator1());
```

The Template Code Generation Pattern

A template contains the structure of generated code.

Tools: Apache Velocity - Java Emitter Templates (JET) - openarchitectureware - Acceleo

```
<?xml version="1.0" encoding="UTF-8"?>
<demo>
<isnice/>
<% for(Iterator i = elementList.iterator();i.hasNext();){ %>
<element><%=i.next().toString()%></element>
<% } %>
<!-- this is part of the generatd XML -->
</demo>
```

See http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html

The Template Code Generation Pattern

```
<%@ jet package="org.jetTest" imports="java.util.List"
class="ComplexGen" %>
<%List<?> objectsToPrint = (List<?>)argument;%>
public class Complex
{
   public void main(String[] args)
   {
<%for (Object objectToPrint : objectsToPrint) {%>
      System.out.println("<%=objectToPrint.toString()%>");
<%}%>
   }
}
```

See http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html

The Interpreter Execution Pattern

- Use the interpreter design pattern
- Every class of the metamodel has a method interpret(Context c)

```
class Context {
  reference currentState : State
  attribute trigger : String
}


class FiniteStateMachine
{
 operation interpret(c : Context) :
Void
 is do
   c.currentState.interpret(c)
 end

}
```

```
class State {
 operation interpret(c : Context):
Void
 is do
   outgoingTransition.each { t |
           t.interpret(c) }
 end
}


class Transition {
 operation interpret(c : Context):Void
  is do
 if c.trigger == self.trigger then
   c.currentState :=  self.target
   stdio.writeln(self.output)
 end
end
}
```

Language: Kermeta. See http://www.kermeta.org.

All the execution semantics is expressed as methods and fields of the metamodel.

```
class FiniteStateMachine
{
  reference currentState : State
  operation execute(events : List<String>) : String is do ... end
}


class State
{
  operation nextState(event : String) : State is do ... end
}


class Transition {
    operation isTriggeredWith(event : String) : Boolean is do
        result := trigger.equals(event)
    end
}
```

Execution semantics: summary of concepts

- Execution Semantics
- The Quick'n'Dirty Code Generation Pattern
- The "toString" Code Generation Pattern
- The Visitor Code Generation Pattern
- The Template Code Generation Pattern
- The Interpreter Execution Pattern
- The Domain Virtual Machine Execution Pattern

Part 4: Static Semantics

Verifying properties

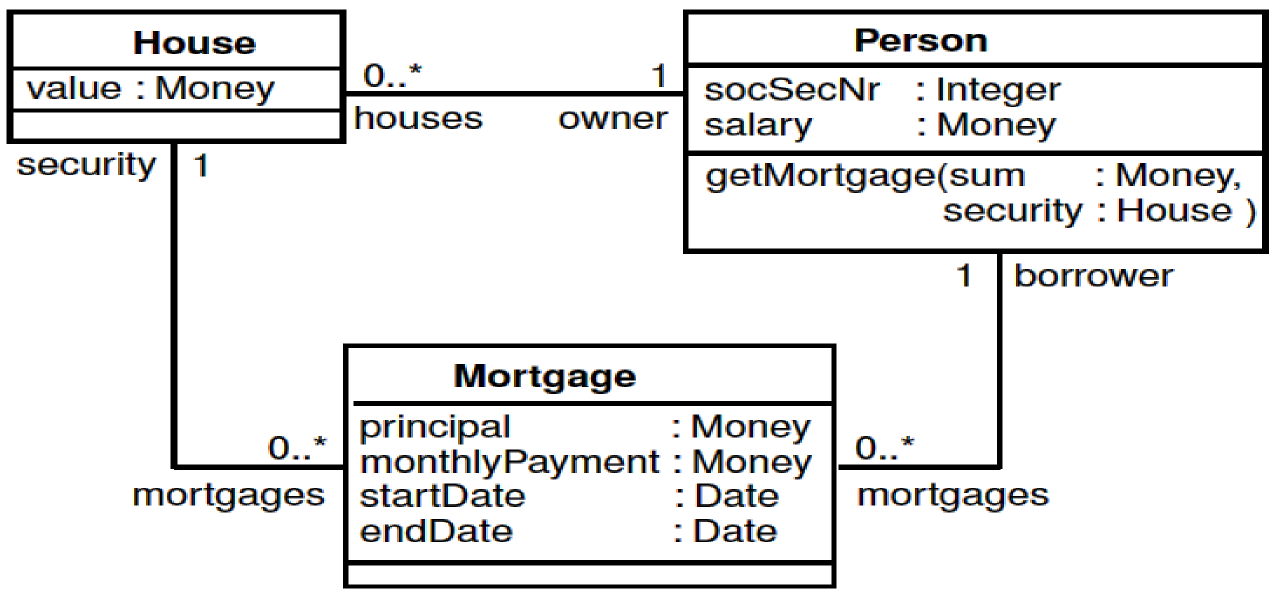
- Static semantics is the description of what are correct and valid models, i.e. structural constraints on models.
- Not all static semantics can be expressed in the metamodel.
- Static semantics can be expressed in a general purpose language, but it's verbose and error-prone.

Example: cycle in inheritance:

```
boolean containsInheritanceCycle(Model m)
  {
    return m.classes.forAll { x|not inheritsFrom(x,x); }
  }

boolean inheritsFrom(Eclass child, EClass parent)
  {
      foreach (Eclass x : child.getEsuperClasses()) {
        if (x==parent) return true;
        if (x.inheritsFrom(parent)) return true;
      }
     return false;
}
```

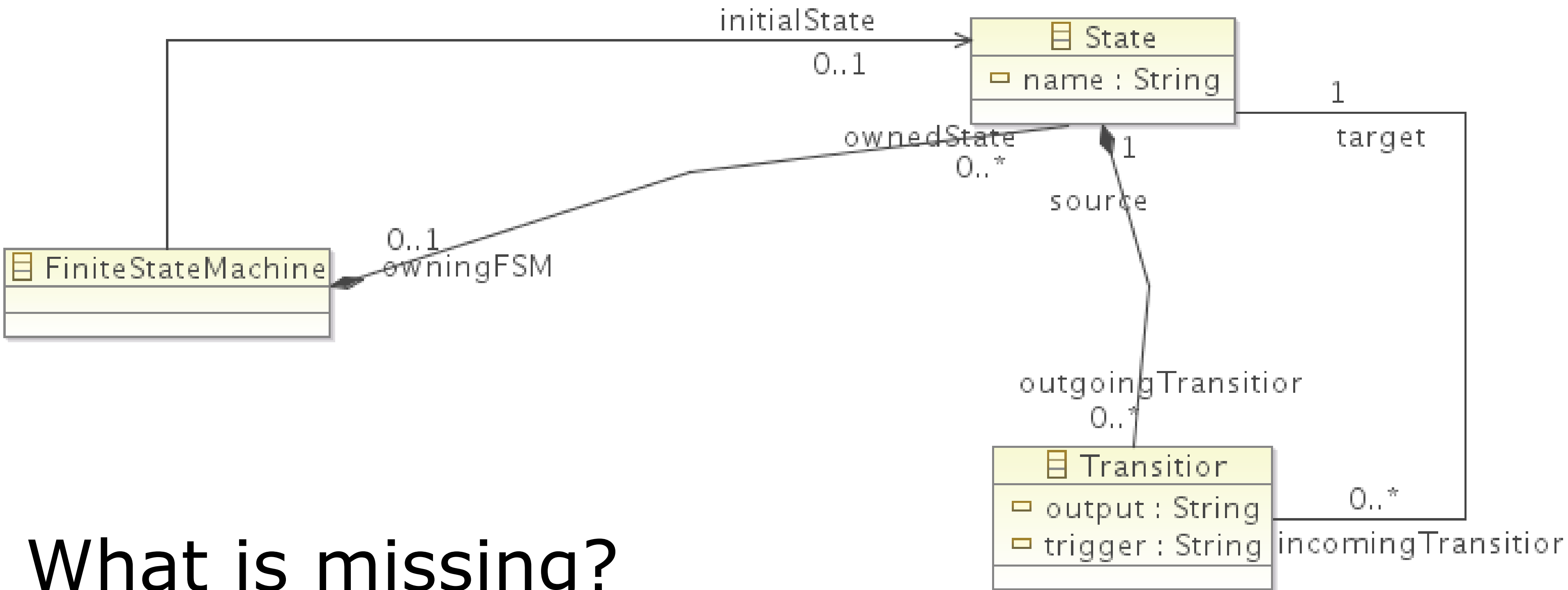
OCL is a language to express static semantics. It can be transformed to Java using different toolkits. (Eclipse MDT-OCL, Dresden OCL Toolkit)



1. **context** *Mortgage*
   invariant: *self.security.owner = self.borrower*

2. **context** *Mortgage*
   invariant: *self.startDate < self.endDate*

Static Semantics for Finite State Machines



What is missing?

-- example of required static semantics
-- determinism invariant
context Transition:
inv: self.source.outGoingTransition->select(x|x.trigger = self.trigger)->size=1

Summary of the lecture

- The biggest thread in model-driven development is to produce cheaper and better systems thanks to code generation.
- Model-driven development can be done with no special tools, and with no graphics.
- Metamodeling languages (e.g. Ecore) provide more powerful modeling constructs than simple OO.
- Metamodeling toolchains (e.g. EMF, Xtext) provides free marshalling and model edition systems.
- We saw 5 different patterns for the execution semantics.
- Static semantics helps developers to create correct models.

Additional reading: The Pragmatics of Model-Driven Development by Bran Selic
http://www.cs.helsinki.fi/u/przybils/courses/CBD06/papers/01231146.pdf