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

Creative Commons Attribution License
Copying and modifying are authorized as long as proper credit is given to the author.
Rennes, Brittany
**Background**

**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 / DRE
École Doctorale : Mathis
Composante universitaire : IPSIC

**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*
François ANDRE
Professeur à l'Université de Rennes 1

* Rapporteurs *
Sébastien DUCASSE
Directeur de Recherche à l'INRIA
Guy SAHRAOUI
Professeur à l'Université de Montréal

*Exameneurs *
Dominique LUDEKNECHT
Directeur à la DGA
Jean-Marc JEZIRIEL
Professeur à l'Université de Rennes 1 (Directeur de thèse)

*Inscrits *
Brigitte HOEFTZERER
Enseignant-chef d'œuvre à l'ENSII
Gilles MARCHALOT
Ingénieur à Thales Airborne Systems

---

**Model-driven generative development of measurement software**

Martin Monperrus · Jean-Marc Jézéquel · Benoît Baudry · Joël Champeau · Brigitte Hoeftzener

Received: 13 October 2009 / Revised: 25 February 2010 / Accepted: 21 April 2010 © Springer-Verlag 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 of applying the approach is evaluated by four case studies. They indicate that this approach significantly eases the measurement activities of model-driven development processes.

1 Introduction
Metrics offer a practical approach [25, 48] to evaluate non-functional properties of artifacts resulting from model-driven engineering (MDE) development processes. Ledezzi 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 certify 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.
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

```c
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

Finite State Machine

Problem

Start

key q /
saveData()

Stop

key x /
loadUI()

Solution

Finite State Machine Spec

Code Generator

C only platform

Expertise? Performance? Reuse?
```python
## 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
```
# 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")
            f.write('}'}
        f.write('while (message !=EOF);'
    if printdebug:
        f.write('printf("Ending...\n");'
    f.write('return 0;}'


# Definition of the model/program

# Imperative manner

```python
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 ****

```python
fsm2C(fsm, 'output.c', False)
```
#define s668 0
#define s546 1
int
main (int argc, char *argv[])
{
    //char* l=argv[1];
    long state = 0;
    char message;
    do
    {
        message = getchar ();
        getchar ();
        switch (state)
        {
        case s668:
            switch (message)
            {
            case 'x':
                state = s546;
                printf ("-> yeahhh\n");
                break;
            default:
                break;
            }
        }
    }
    ...
}
# 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
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

```python
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.

\[
\text{fsm2C(fsm2,'output2.c',False)}
\]

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

\[
\text{Istates:}
- !<\text{state}>
  \text{name: s74}
\]

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

See: #Slide 7
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="Dictionary"/>
</library:Library>
```
Concept: multiplicity (EMOF)

I can forget this!

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

```xml
!<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

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

Serialization

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

\[
\text{!fsm>
\begin{align*}
\text{!states:} \\
\text{- !state>}
\text{name: s74} \\
\text{transitions:} \\
\text{- !transition>}
\text{trigger: z} \\
\text{message: b} \\
\text{tostate: s75} \\
\text{- !state>}
\text{name: s75} \\
\text{transitions:} \\
\text{- !transition>}
\text{trigger: l} \\
\text{message: a} \\
\text{tostate: s74}
\end{align*}
\]
\]

\[
\text{!fsm>
\begin{align*}
\text{!states:} \\
\text{- &id001 !state>}
\text{name: s966} \\
\text{transitions:} \\
\text{- !transition>}
\text{message: yeahhh} \\
\text{tostate: &id002 !state>}
\text{name: s888} \\
\text{transitions:} \\
\text{- !transition>}
\text{message: cool} \\
\text{tostate: *id001} \\
\text{trigger: y} \\
\text{trigger: x} \\
\text{tostate: s74}
\end{align*}
\]
\]

- *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)
The Quick'n'Dirty Code Generation Pattern

```python
# 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) + '
')
    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) + ';
            f.write('printf("-> ' + t.message + ")
            f.write('}
    while (message != EOF);
    if printdebug:
        f.write('printf("Ending...
        f.write('return 0;
```
The "toString" Code Generation Pattern

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

```java
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+"\n State currentState \n}"
}
class LeafState {
    String toCode() {
        // the generated code uses the State pattern as State
        return "class "+this.name+"extends State {}\n}"
}
```
The Visitor Code Generation Pattern

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

```java
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 generated XML -->
</demo>

See http://www.eclipse.org/articles/Article-JET/jet_tutorial11.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()%>");
        <%}%>
    }
}
The Interpreter Execution Pattern

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

```kermeta
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
            stdout.writeln(self.output)
        end
    end
}
```

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:
```java
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
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