Model-Driven Language Engineering (Part 2)

Franck Fleurey
e-mail: Franck.Fleurey@sintef.no
http://www.fleurey.com/franck

SINTEF
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 familiar 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.kmt”
// Transformations

class Minimizer {
    operation minimize (source: FSM):FSM {...}
}
Kermeta: a **Kernel metamodeling 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.
EMOF ↔ Kermetta

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 do
    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 do
    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

Before

A

b

0..1

B

After

a2. b := b1

Association

Before

C

c

1

D

d

After

d2. c := c1

Avant

Après

Avant

Après

Composition Association

Before

A

b

container()

B

a1:A

b1:B

After

C

c

1

D

d

c1:C

d1:D

d2:D

d2. c := c1

After

C

c

1

D

d

c1:C

d1:D

d2:D

d2. c := c1

Avant

Après

Avant

Après
Example

**operation** fire() : String

source.owningFSM.currentState := target

result := output
operation step(c : String) : String

// Get the valid transitions
var validTransitions : Collection<Transition>
validTransitions := outgoingTransition.select { t |
  t.input.equals(c)
}

// Check if there is one and only one valid transition
if validTransitions.empty then raise NoTransition.new end
if validTransitions.size > 1 then
  raise NonDetermism.new
end

// fire the transition
result := validTransitions.one.fire
operation run() : Void

from var str : String
until str == "exit"
loop
    stdio.writeln("current state is " + currentState.name)
    str := stdio.read("Enter an input string or 'exit' to exit simulation : ")

    stdio.writeln(str)
    if str != "exit" then
        do
            stdio.writeln("Output string : " + currentState.step(str))
        rescue (ex : FSMException)
            stdio.writeln("ERROR : " + ex.toString)
        end
    end
end
stdio.writeln("* END OF SIMULATION *")
/**
 * 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

- Defined as an Ecore meta-model
require "VMLogo.ecore"
require "TurtleGUI.kmt"

aspect class Point {
  method toString() : String is do
    result := "[" + x.toString + "," + y.toString + "]"
  end
}

aspect class Turtle {
  operation setPenUp(b : Boolean) is do
    penUp := b
  end
  operation rotate(angle : Integer) is do
    heading := (heading + angle).mod(360)
  end
}
Map Instructions to VM Actions

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

```java
aspect class PenUp {
    eval (ctx: Context) {
        ctx.getTurtle().setPenUp(true)
    }
}
...
aspect class Clear {
    eval (ctx: Context) {
        ctx.getTurtle().reset()
    }
}
```
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
Handling control structures

- Block
- Conditional
- Repeat
- While
Operational semantics

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

aspect class If {
    operation eval(context : Context) : Integer is do
        if condition.eval(context) != 0 then
            result := thenPart.eval(context)
        else result := elsePart.eval(context)
    end
end

aspect class Right {
    operation eval(context : Context) : Integer is do
        context.turtle.rotate(angle.eval(context))
    end
}
```

LogoDynSemantics.kmt
Handling function calls

- Use a stack frame
  - Owned in the Context
Getting an Interpreter

- Glue that is needed to load models
  - i.e. LOGO programs

- Visualize the result
  - Print traces as text
  - Put an observer on the LOGO VM to graphically display the resulting figure
Simulator

- Execute the operational semantics

```plaintext
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
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

```java
aspect class PenUp {
    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
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
Classification of Model-to-Model Transformation Techniques

1. General purpose programming languages
   - Java/C#...

2. Generic transformation tools
   - Graph transformations, XSLT...

3. CASE tools scripting languages
   - Objecteering, Rose...

4. Dedicated model transformation tools
   - OMG QVT style

5. Meta-modeling tools
   - Metacase, Xactium, Kermeta...
Logo to NXC Compiler

- **Step 1 – Model-to-Model transformation**
  - ASMLogo.ecore
  - K.xmi
  - Logo4NXC.kmt
  - K1.xmi
  - ASMLogo.ecore

- **Step 2 – Code generation with template**
  - ASMLogo.ecore
  - K1.xmi
  - NXC.ket
  - K.nxc
  - ASMLogo.ecore
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 {
  method local2global(rootBlock: Block) is do
    end
}
aspect class ProcDeclaration
  method local2global(rootBlock: Block) is do
    ...
    end
}
aspect class Block
  method local2global(rootBlock: Block) is do
    ...
    end
}
```

Step 2: Kermeta Emitter Template

- NXC Code generation using a template
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)

- ASMLogo.ecore
- TurtleGUI.java
- VMLLogo.ecore
- TurtleGUI.kmt
- TurtleControler.kmt
- LogoVMSemantics.kmt
- LogoDynSemantics.kmt
- LogoSimulator.kmt
- LogoNCXCompiler.kmt
- Editor.java
- logo.sts
- NXC.ket
- LogoStaticSem.ocl
- Logo4NXC.kmt

Diagram showing dependencies between various files and packages.
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

Interaction between the current simulation (Kermeta) and the GUI (Java)

Static constraints in OCL

Transformation written in Kermeta

Embedded source code inside the robot

Result of a real execution

Input scenario

Logo Semantic in Kermeta

Result of a simulation interpreted with Kermeta

Transformation written in Kermeta
Kermeta in real projects

- Artist2, the European network of excellence on real-time embedded systems
- UsineLogicielle, a System@tic project where Kermeta based operational semantic is associated to functional requirement for test synthesis purposes.
- Speeds, a European FP6 project for aspect-oriented metamodeling of avionics and automotive systems, including operational semantics aspects
- OpenEmbedd, A French project building a MDE platform for realtime system.
- Mopcom, a French project applying MDE to hardware for generating SOC and introduce dynamic reconfigurability to them.
- Topcased, a consortium that aims to build modelers for avionics and system engineering
- DiVA, a European FP7 STREP project on handling Dynamic variability in complex, adaptive systems
Conclusion and Wrap-up

- Kermeta is an open-source initiative
  - Started January 2005
  - More than 10 active developers

- Feel free to use
  - Start with a meta-model in EMF
    » Get XML an reflective editor for free
  - Weave in static semantics in OCL
  - Weave in an interpreter,
    » connect to a simulation platform
  - Weave in one or more compilers
  - Finally care for concrete syntax issues

- Feel free to contribute!
  - www.kermeta.org
**OBLIG 1 a/b**

1a. Training in EMF and EMF Validation Framework

The goal of this exercise is to create a meta-model for the LOGO language and to implement its static semantics using the EMF Validation Framework.

You can choose any version of the logo language you want to implement (see [http://en.wikipedia.org/wiki/Logo_(programming_language)](http://en.wikipedia.org/wiki/Logo_(programming_language))) but it should have at least procedure definitions and calls in addition to the basic constructions.

1b. Training in EMFText

The EMFText tool allow to specify textual concrete syntax for domain specific modelling languages. The eclipse plugin, documentation and many examples can be found at [http://www.emftext.org](http://www.emftext.org).

The goal of the exercise is to create a text editor for your LOGO language. Your textual syntax should include proper parsing of expressions with correct priorities of operators and without the need for redundant parenthesis. Hint: make use of the operators annotations available in the latest version of emftext. Your parser should ensure proper scoping of variables and especially procedure parameters.
OBLIG 1c

This part can be done in groups. The goal is to make the LOGO language which you have built executable either by interpretation or compilation. Each group can choose between the tools:

- Kermeta: can be used to build either an interpreter or a compiler.
- MOFScript: More suited to build a compiler.
- Java: More suited to build an interpreter.
- The result should allow executing the LOGO program and visualizing the result. The visualization can be implemented as a simple JAVA GUI.