OpenEmbeDD lectures

Kermeta 1.2

Advanced use

Contents

- « Model checking » with *Kermeta*
- Transformating models
- Simulating models
- *[TODO] use of reflexivity*
- . . .

More information : http://kermeta.org/documents/

Chapter One

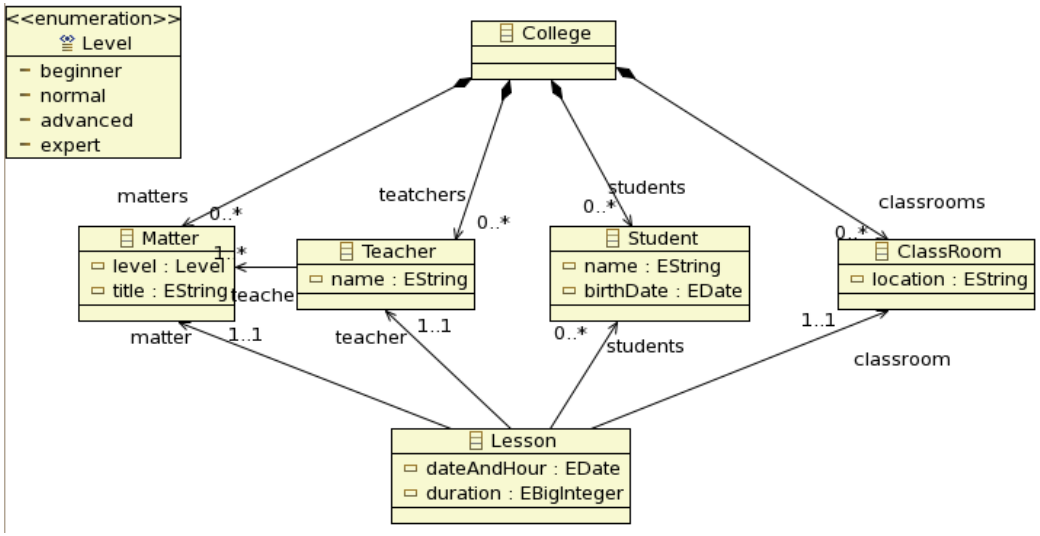
« Model checking » with *Kermeta*

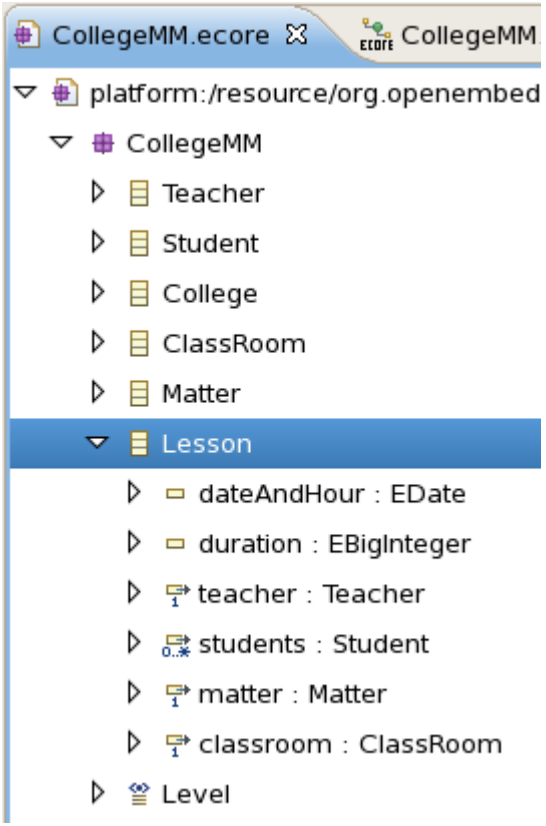
I – Checking: constraints in Kermeta

- **Kermeta contracts on models**
  - `pre`, `post` & `inv` aspects contracts on metamodel
  - Static verifications: `inv` constraints
  - Verifications on running: `pre` & `post` constraints
- **OCL constraints and Kermeta**
  - Write OCL constraints in an `.ocl` file
  - Use OCL->Kermeta transformation
  - Require `.kmt` file as Kermeta contract

I – Checking: school use case

- **College example metamodel**





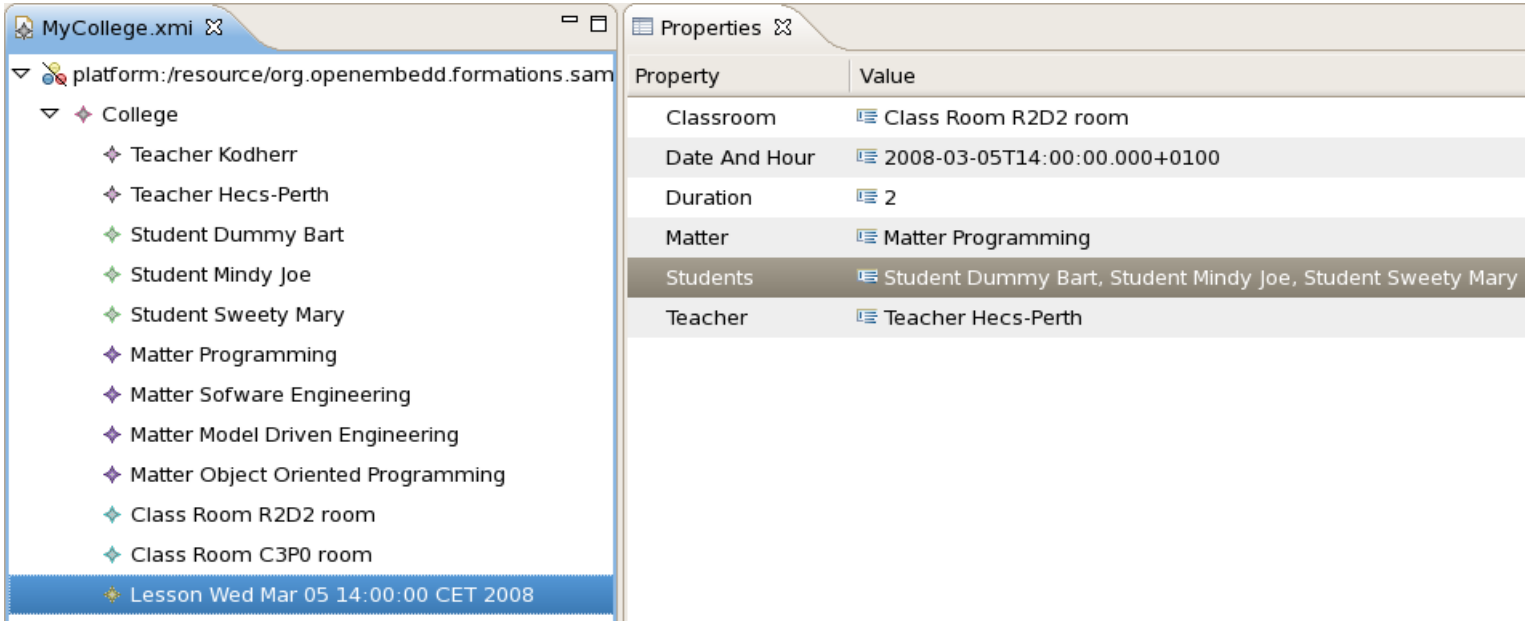
I – Checking: constraints in Kermeta

- **Define constraints on a metamodel**

```
@mainClass "CollegeMM::Verification"
@mainOperation "main"

package CollegeMM;

require kermeta
require "platform:/resource/org.openembedd.formations.samples.kermeta02/metamodel/CollegeMM.ecore"

aspect class Lesson {
   inv lessonHasTeacher is do
      self.teacher != void
   end
   inv lessonHasStudents is do
      self.students.size > 1
   end
   inv lessonHasMatter is do
      self.matter != void
   end
   inv lessonHasClassroom is do
      self.classroom != void
   end
}



class Verification {
   operation main() : Void is do
      var rep : kermeta::persistence::Repository init kermeta::persistence::EMFRepository.new
      var res : kermeta::persistence::Resource init rep.getResource(
                "platform:/resource/org.openembedd.formations.samples.kermeta02/model/MyCollege.xmi")
      var model : CollegeMM::College
      model ?= res.one
      model.checkAllInvariants
   end
}
```

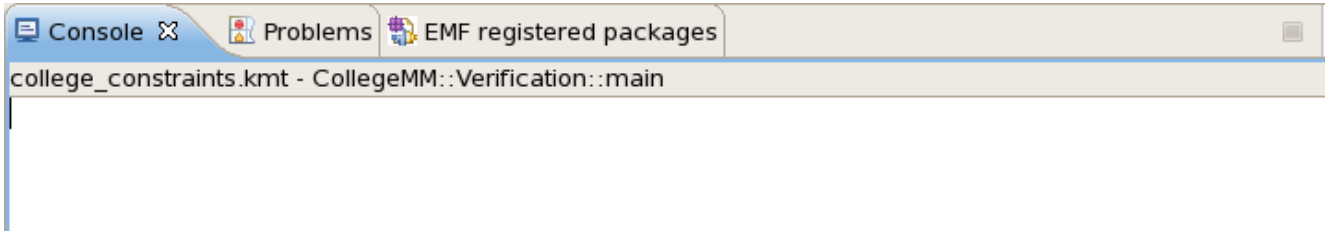
I – Checking: constraints in Kermeta

- **A correct model to test: MyCollege.xmi**



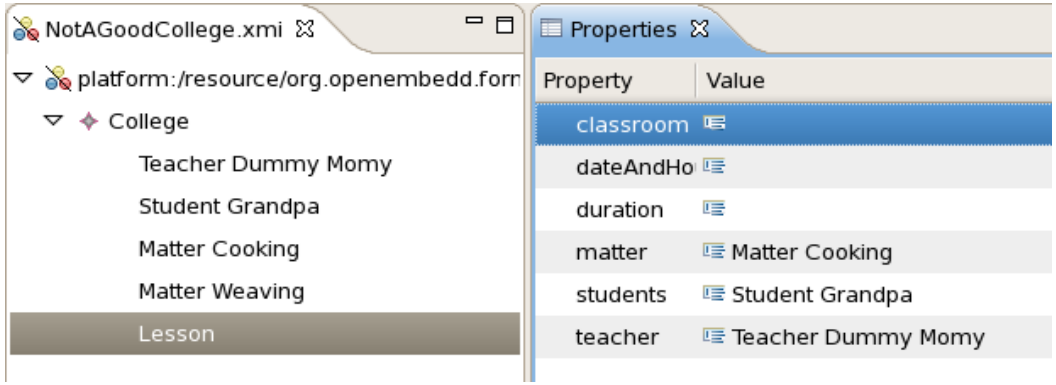
The lesson has all its references ...

... so the verification runs without errors



I – Checking: constraints in Kermeta

- **A bad model to test: NotAGoodCollege.xmi**



The lesson lacks some of its references ...

... so the verification raises errors

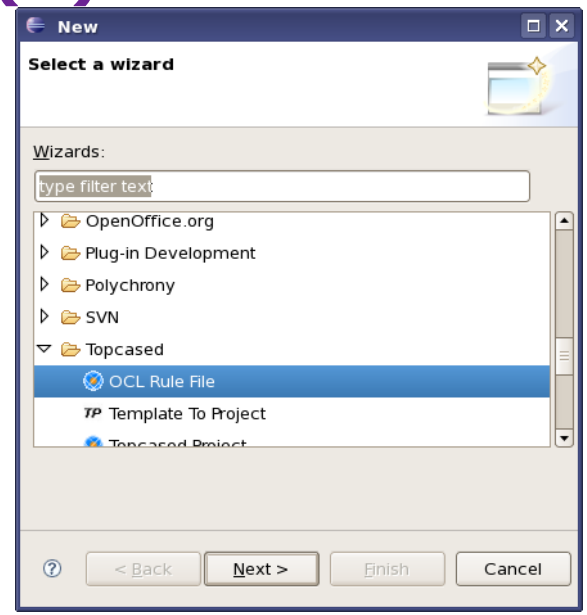
Console | Problems | EMF registered packages

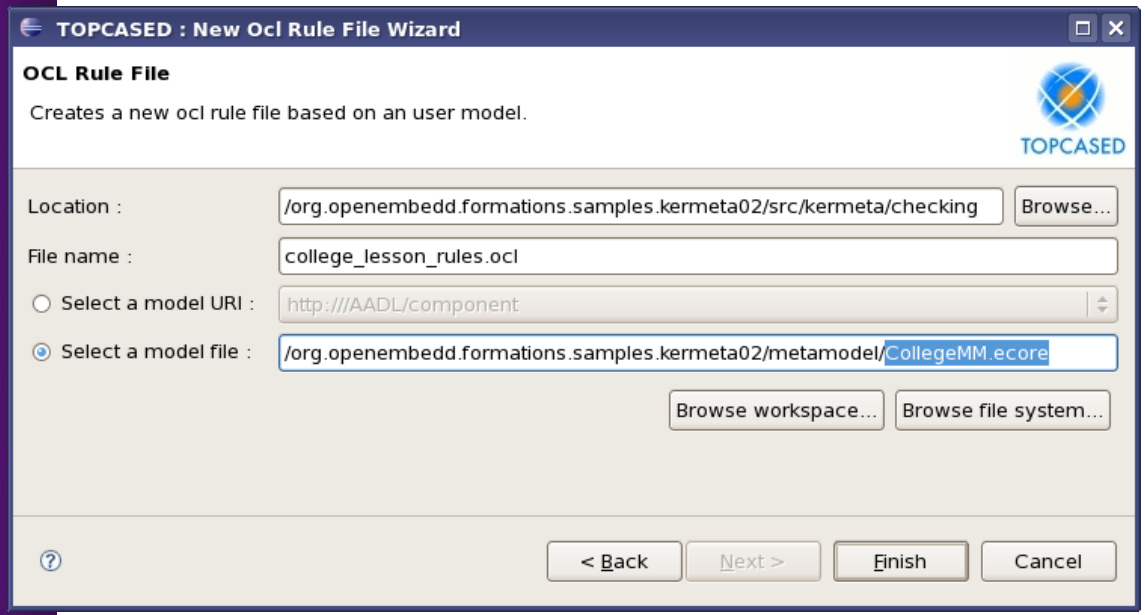
college_constraints.kmt - CollegeMM::Verification::main

```
kermeta exception : [kermeta::exceptions::ConstraintViolatedInv : 10894]
fr.irisa.triskell.kermeta.interpreter.KermetaRaisedException: kermeta exception : [ke
Inv lessonHasStudents of class Lesson violated
Trace:
      [CollegeMM::Lesson : 8229].checkInvariants
     [CollegeMM::Lesson : 8229].checkAllInvariants
     [CollegeMM::College : 8216].checkAllInvariants#function call
     [kermeta::language::ReflectiveSequence : 10616 = "[[CollegeMM::Lesson : 8229]]"]
     [CollegeMM::College : 8216].checkAllInvariants#function call
     [kermeta::language::ReflectiveSequence : 10791 = "[[kermeta::language::structure
     [CollegeMM::College : 8216].checkAllInvariants
     [CollegeMM::Verification : 8094].main
------------END OF STACK TRACE------------
```

I – Checking: constraints in OCL

- **Write the constraints in OCL (1)**





I – Checking: constraints in OCL

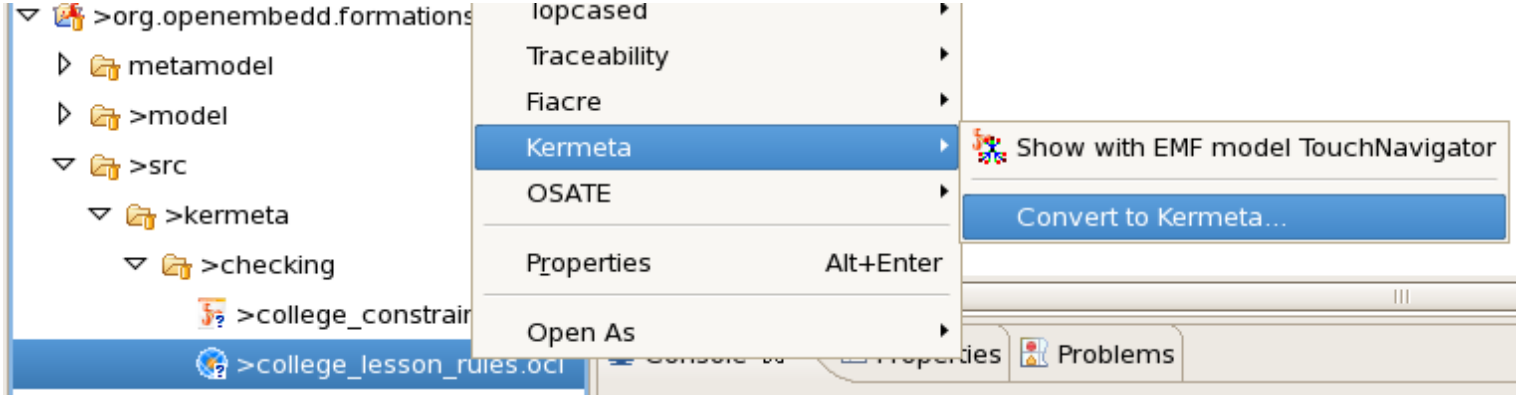
- **Write the constraints in OCL (2)**

college_lesson_rules.ocl

```
MainModel : /org.openembedd.formations.samples.kermeta02/metamodel/CollegeMM.ecore

context Lesson

inv lessonHasTeacher:
    self.teacher->notEmpty()

inv lessonHasStudents:
    self.students->size() > 1

inv lessonHasMatter:
    self.matter->notEmpty()

inv lessonHasClassroom:
    classroom->notEmpty()
```

I – Checking: constraints in OCL

- **Change the OCL into Kermeta**



Transforming models

II – Transformation: goal

- **An instance of a metamodel as input**
  - For example: a UML2 state machine
- **Another metamodel as target**
  - The Kermeta FSM corresponding instance
- **An automated process to produce the second**
  - First model as input
  - Our Kermeta code
  - Second model as output

Many other transformations than One-to-One can be realized, due to Kermeta ability to *compute* models

II – Transformation : solution 1 (Visitor)

The *Visitor* pattern gives us the ability to navigate the input model and execute action(s) on each of its elements

- An `accept(Visitor)` method on existing classes
- Any number of concrete visitors, for tasks to be done
- A `visit(X)` method in the visitor for each metamodel element accepting X

Create references implies the relied instances must have been created

- First pass : instantiate objects (first concrete visitor)
- Second pass : rely them (second concrete visitor)

  The output instances build leads to a *Builder* pattern

II – Transformation : solution 2 (aspects)

Kermeta "aspects" give us the ability to extend the metamodel behaviour as we need

- Add on each element
  - A first pass to create output element
  - A second pass to link output elements together
  - A browsing code to apply passes on sub-elements

The tree of UML2 input model

- uml::Model
  - uml::Package
    - uml::StateMachine
      - uml::Region
        - uml::Vertex
        - uml::Transition

II – Transformation : solution 2 (aspects)

Browser: navigate through input model

```
package uml;

require kermeta
require "http://www.eclipse.org/uml2/2.1.0/UML"

aspect class Element {
operation eachOwnedElement(func : <Element -> Element>)
              : Void is abstract
}

aspect class Model {
   method eachOwnedElement(func : <Element -> Element>)
              : Void is do
      self.packagedElement.each { o |
         func(o)
      }
   end
}

aspect class Package {
   method eachOwnedElement(func : <Element -> Element>)
              : Void is do
      self.packagedElement.each { o |
         func(o)
      }
   end
}
```

```
aspect class StateMachine {
   method eachOwnedElement(func : <Element -> Element>)
              : Void is do
      self.region.each{ o |
         func(o)
      }
   end
}

aspect class Region {
   method eachOwnedElement(func : <Element -> Element>)
              : Void is do
      self.subvertex.each{ o |
         func(o)
      }
      self.transition.each{ o |
         func(o)
      }
   end
}
```

II – Transformation : solution 2 (aspects)

First pass: build target entities

```
package uml;

require kermeta
require "UmlBrowser.kmt"
require "http://www.kermeta.org/fsm"

// the main class in UML2 (all other classes derived from it)
aspect class Element {
   operation uml2fsmPass1() is abstract
}

aspect class Model {
   method uml2fsmPass1() is do
      self.eachOwnedElement{ p | p.uml2fsmPass1() } // browse
   end
}

aspect class Package {
   method uml2fsmPass1() is do
      self.eachOwnedElement{ p | p.uml2fsmPass1() } // browse
   end
}

aspect class StateMachine {
   reference output : fsm::FSM
   method uml2fsmPass1() is do
      output := fsm::FSM.new
      self.eachOwnedElement{ p | p.uml2fsmPass1() } // browse
   end
}
```

```
aspect class Region {
   reference outModel : fsm::FSM
   method uml2fsmPass1() is do
      self.eachOwnedElement{ p | p.uml2fsmPass1() }
// browse
   end
}

aspect class Vertex {
   reference output : fsm::State
   reference outModel : fsm::FSM
   method uml2fsmPass1() is do
      output := fsm::State.new
      output.name := self.name
   end
}

aspect class Transition {
            // nothing to do as in FSM transitions
are links
}
```

II – Transformation : solution 2 (aspects)

Second pass: link target entities together

```
package uml;

require kermeta
require "UmlBrowser.kmt"
require "http://www.kermeta.org/fsm"

// the main class in UML2 (all other classes derived from it)
aspect class Element {
   operation uml2fsmPass2() is abstract
}

aspect class Model {
   method uml2fsmPass2() is do
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}

aspect class Package {
   method uml2fsmPass2() is do
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}

aspect class StateMachine {
   reference output : fsm::FSM
   method uml2fsmPass2() is do
      /* the region does not know directly its state machine
         so we must pass it to the Pass2 method */
      self.region.each{ r | r.outModel := self.output }
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}
```

```
aspect class Region {
   reference outModel : fsm::FSM
   method uml2fsmPass2() is do
      self.subvertex.each{ sv |
         sv.outModel := outModel
         outModel.ownedState.add(sv.output)
      }
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}

aspect class Vertex {
   reference output : fsm::State
   reference outModel : fsm::FSM
   method uml2fsmPass2() is do
      outModel.ownedState.add(output)
      var pseudoState : uml::Pseudostate
      pseudoState ?= self
      if pseudoState != void then
         if pseudoState.kind ==
               uml::PseudostateKind.initial then
            outModel.initialState := output
         end
      end
      var finalState : uml::FinalState
      finalState ?= self
      if finalState != void then
         outModel.finalState.add(output)
      end
   end
}

aspect class Transition {
   // nothing to do as in FSM transitions are links
}
```

II – Transformation: 2 (aspects)

Automated generation of the browser

- Copy the metamodel (xx.ecore) in local workspace
- Get the Kermeta MDK for Ecore
  - Fetch *fr.irisa.triskell.kermeta.ecore* from Kermeta CVS
  - In *src/kermeta/transformations*
    - Launch *ContainmentBasedActionPerformerGenerator.kmt*
      - With the local metamodel as parameter

Obtain the browsing code

- In local workspace
  - *xx.ecore_ContainmentBasedActionPerformer.kmt*
- Rename it
  - *xx_browser.kmt*

You can now require it in your tasks code

Simulating models

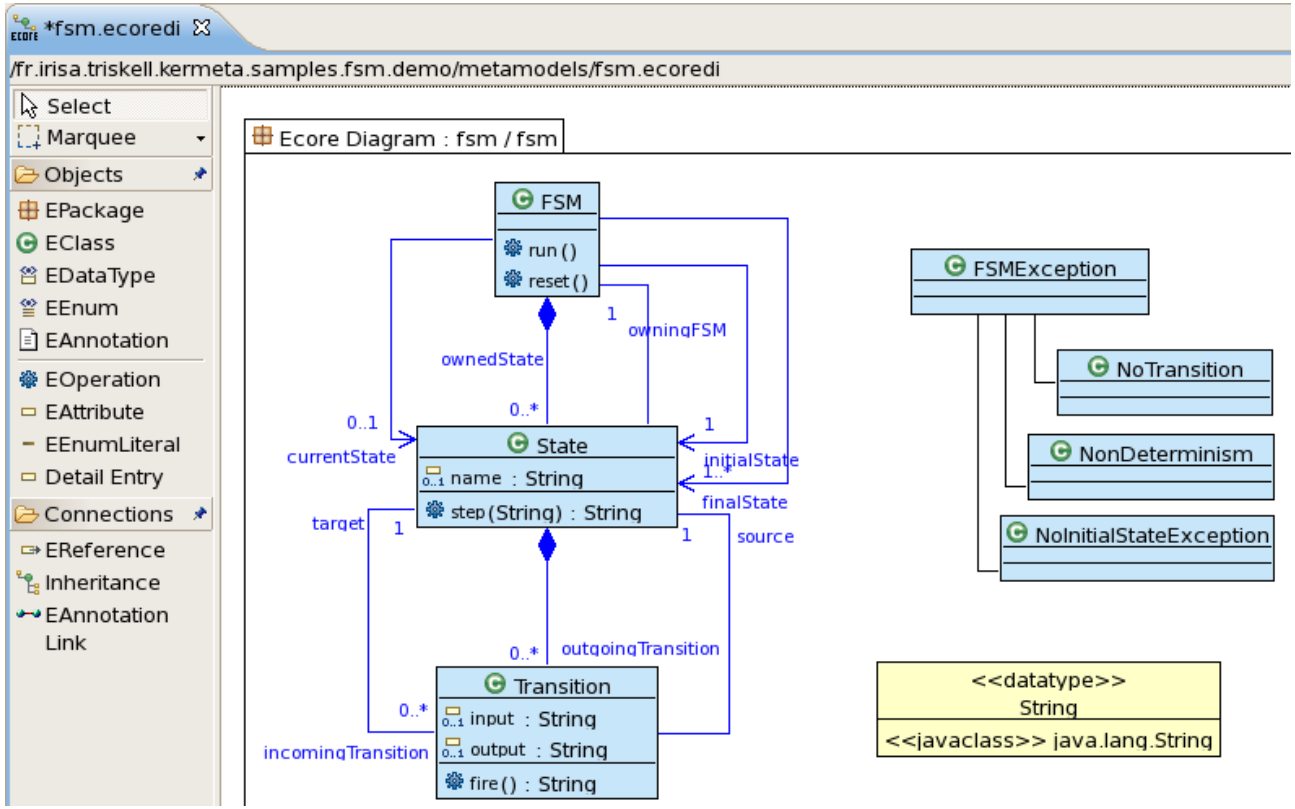
III – Simulation : principles

- **Execute a model means**
  - add behaviour to the metamodel
  - input data from the *Actor* who interacts with model
  - display the current state of the model under simulation
  - run the whole thing in an *Actor* (user) interface
- **Kermeta feeds all theses needs**
  - Aspects can define the code of operations
  - The Kermeta console can read strings
  - The Kermeta console can display strings
  - The Kermeta launcher can easily run *.km* or *.kmt* files

III – Simulation : the FSM example

- **The FSM metamodel**

  (see FSM tutorials inside Kermeta distribution)



III – Simulation : the FSM example

- **The FSM has three behaviour points**
  - **State** class `step(String) : String` method
    - Get the external input
    - Select another state depending on the input value
    - Return the output related to the involved transition
  - **Transition** class `fire() : String` method
    - Realize the action corresponding to the transition
    - Change the current state of the machine
    - Return the produced String
  - **FSM** class `run()` method
    - Plug the model instance into a simulator
    - Initialize the corresponding machine
    - Launch the whole execution process

III – Simulation : the FSM example

- **Code part 1**

```
@mainClass "fsm::Simulator"
@mainOperation "main"

package fsm;

require kermeta
require "http://www.kermeta.org/fsm"

using kermeta::standard

aspect class State {
   operation step(c : String) : String raises FSMException is do
      var validTransitions : Collection<Transition>
      validTransitions := outgoingTransition.select{ t | t.input.equals(c) }

      if validTransitions.empty then raise NoTransition.new end
      if validTransitions.size > 1 then raise NonDeterminism.new end

      result := validTransitions.one.fire
   end
}

aspect class Transition {
   operation fire() : String raises FSMException is do
      source.owningFSM.currentState := target
      result := output
      stdio.writeln(" --> "+result)
   end
}
```

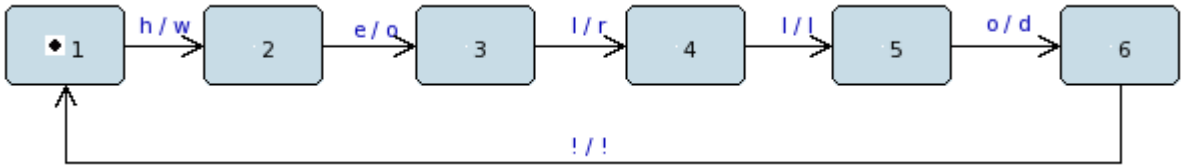
III – Simulation : the FSM example

- **Code part 2**

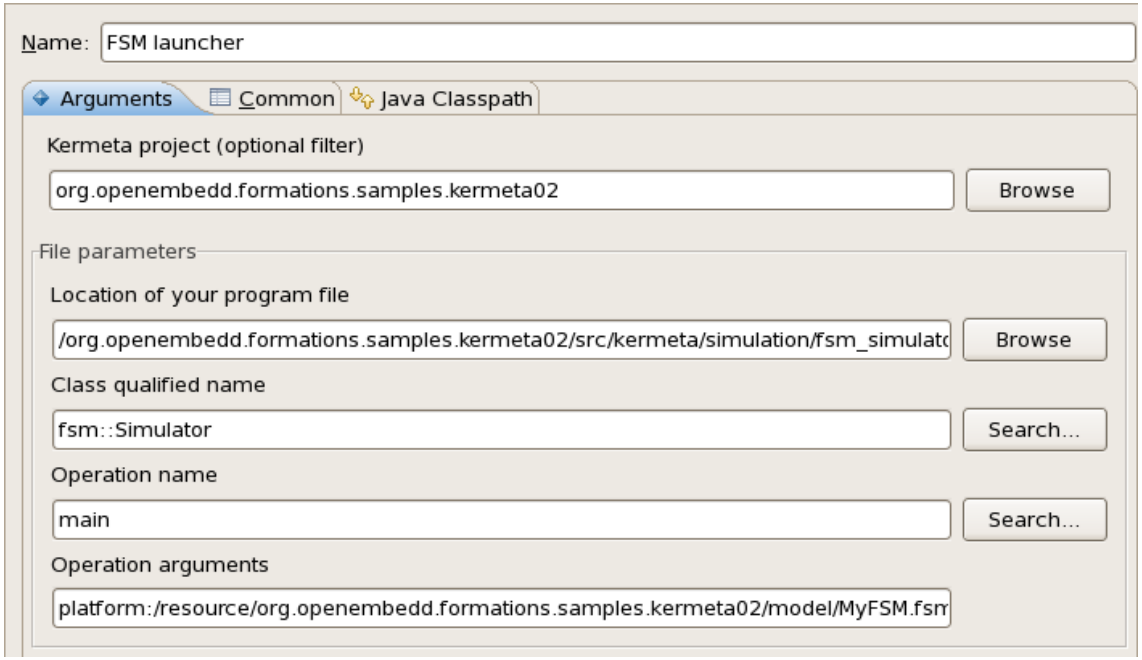
```
aspect class FSM {
   operation run() : Void raises FSMException is do
      from var str : String init "init"
      until str == "quit"
      loop
         if( str != "init") then
            do
               var res : String
               res := self.currentState.step(str)
            rescue (err : NoTransition)
               stdio.writeln(err.toString)
            rescue (err : NonDeterminism)
               stdio.writeln(err.toString)
            end
         end
         stdio.writeln("Current state: " + self.currentState.name)
         str := stdio.read(" give me a string > ")
      end
   end
}

class Simulator {
   operation main(modelFile : String) : Void is do
      var rep : EMFRepository init EMFRepository.new
      var res : Resource init rep.getResource(modelFile)
      var model : FSM
      model ?= res.one
      model.currentState := model.initialState
      model.run
      stdio.writeln(" -----> exiting!")
   end
}
```

III – Simulation : the FSM example

- **A model to execute**

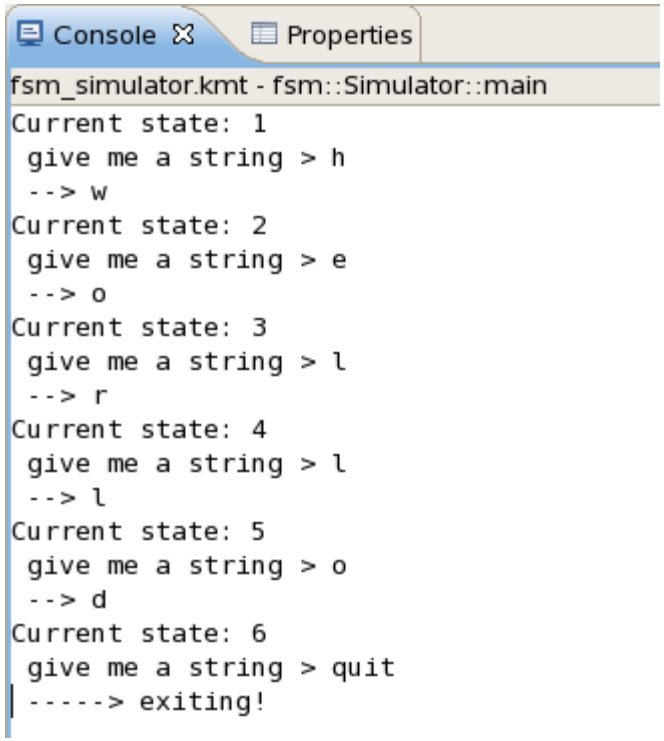


- **The execution parameters**



III – Simulation : the FSM example

- **The trace of an execution**



OpenEmbeDD lectures

Kermeta 1.2

Advanced use

Contents

- « Model checking » with *Kermeta*
- Transformating models
- Simulating models
- *[TODO] use of reflexivity*
- . . .

More information : http://kermeta.org/documents/

This course is adressed to people familiar with Kermeta features.

We present advanced principles, based upon Kermeta, for main MDE relative tasks.

The list will be extended in the future.

The Aspects oriented approach makes manipulation of models easier through a natural writing of processes onto the metamodel itself. Aspects infer the models behave as desired, their elements doing themselves the changes.

Chapter One

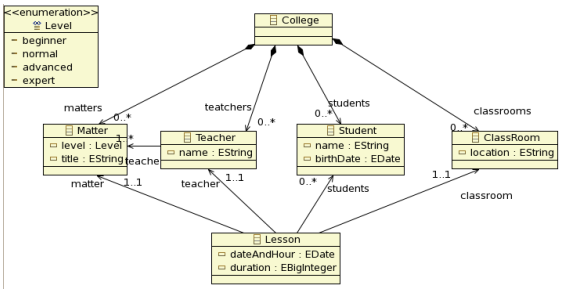
« Model checking »
with *Kermeta*

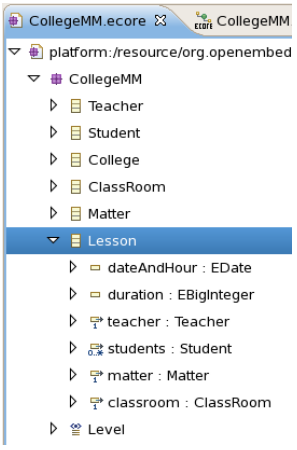
I – Checking: constraints in Kermeta

- **Kermeta contracts on models**
  - `pre`, `post` & `inv` aspects contracts on metamodel
  - Static verifications: `inv` constraints
  - Verifications on running: `pre` & `post` constraints
- **OCL constraints and Kermeta**
  - Write OCL constraints in an `.ocl` file
  - Use OCL->Kermeta transformation
  - Require `.kmt` file as Kermeta contract

I – Checking: school use case

- **College example metamodel**





In order to make instances creation easier, you must define a root container and containers for every element of the Ecore metamodel.

Then you will be able to "create dynamic instance" model from the metamodel on the "College" root and "Add child" to it.

I – Checking: constraints in Kermeta

- **Define constraints on a metamodel**

```
@mainClass "CollegeMM::Verification"
@mainOperation "main"

package CollegeMM;

require kermeta
require "platform:/resource/org.openembedd.formations.samples.kermeta02/metamodel/CollegeMM.ecore"

aspect class Lesson {
   inv lessonHasTeacher is do
      self.teacher != void
   end
   inv lessonHasStudents is do
      self.students.size > 1
   end
   inv lessonHasMatter is do
      self.matter != void
   end
   inv lessonHasClassroom is do
      self.classroom != void
   end
}


class Verification {
   operation main() : Void is do
      var rep : kermeta::persistence::Repository init kermeta::persistence::EMFRepository.new
      var res : kermeta::persistence::Resource init rep.getResource(
                "platform:/resource/org.openembedd.formations.samples.kermeta02/model/MyCollege.xmi")
      var model : CollegeMM::College
      model ?= res.one
      model.checkAllInvariants
   end
}
```

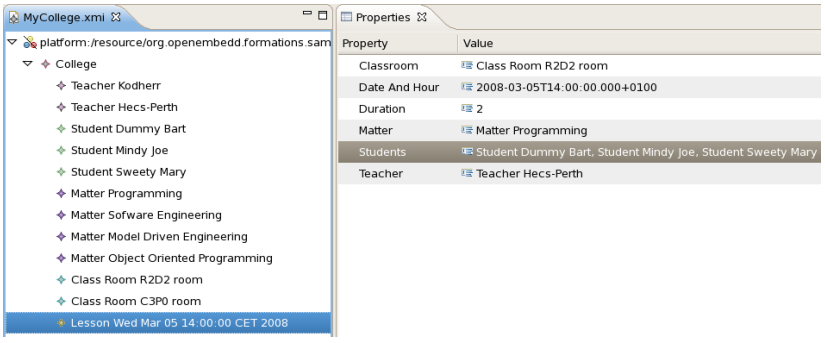

As you want to add contracts with aspects on existing elements of the metamodel, you must choice the metamodel package as the root package of your .kmt file.

The “Verification” class could be in another Kermeta file, as it is a launcher.

We only present invariants here. See first level lecture of Kermeta language for other features of Kermeta contracts.

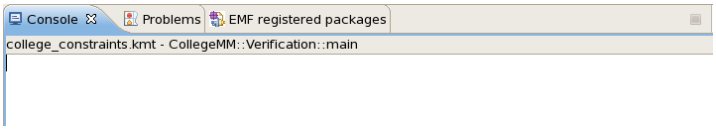
I – Checking: constraints in Kermeta

- **A correct model to test: MyCollege.xmi**



The lesson has all its references ...

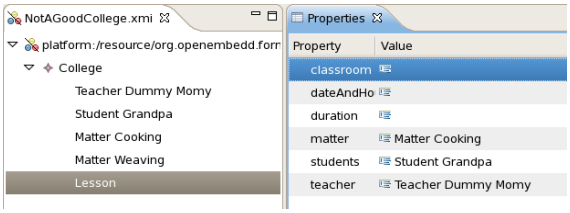
... so the verification runs without errors



Create the model as it is presented (you can use different names :-))

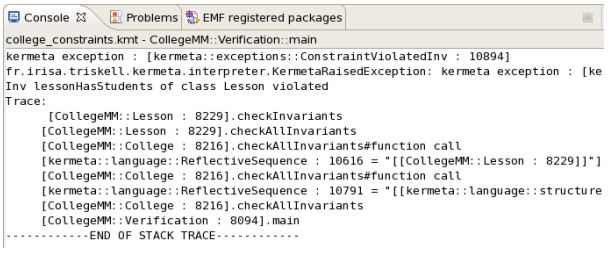
I – Checking: constraints in Kermeta

- **A bad model to test: NotAGoodCollege.xmi**



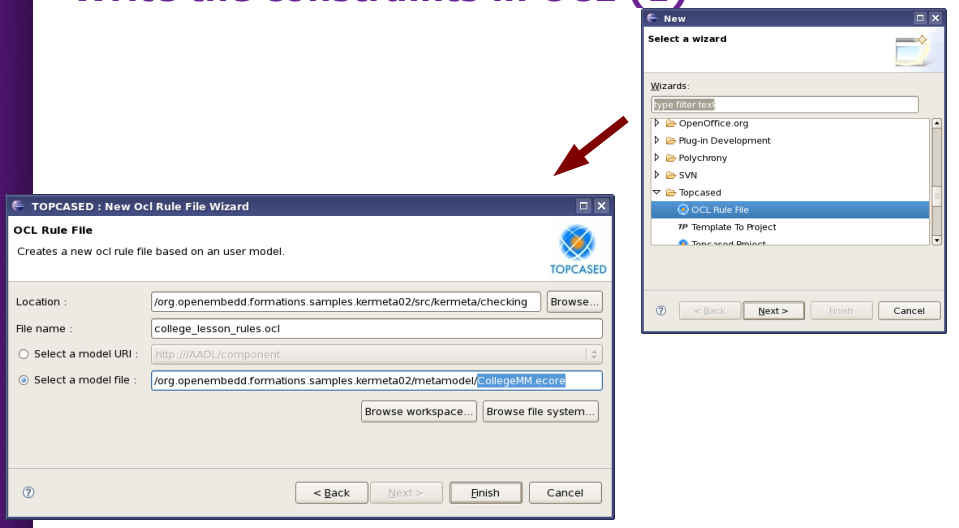
The lesson lacks some of its references ...

... so the verification raises errors



I – Checking: constraints in OCL

- **Write the constraints in OCL (1)**



We use the Topcased OCL editor, which offers completion and other functions we can expect from a professional editor.

I – Checking: constraints in OCL

- **Write the constraints in OCL (2)**

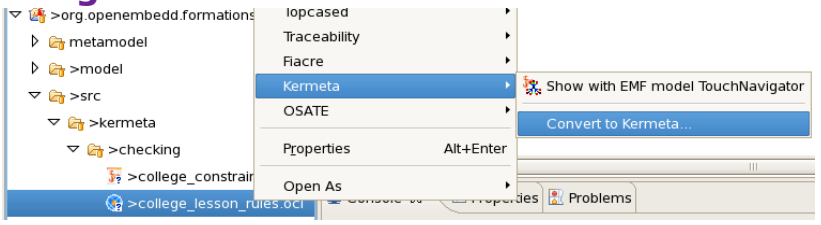
college_lesson_rules.ocl

```
MainModel : /org.openembedd.formations.samples.kermeta02/metamodel/CollegeMM.ecore

context Lesson

inv lessonHasTeacher:
    self.teacher->notEmpty()

inv lessonHasStudents:
    self.students->size() > 1

inv lessonHasMatter:
    self.matter->notEmpty()

inv lessonHasClassroom:
    classroom->notEmpty()
```


Have a look on slide 6 and see how close are Kermeta and OCL syntax when navigating in collections.

I – Checking: constraints in OCL

- **Change the OCL into Kermeta**



The export OCL -> Kermeta may not be as stable as we expect for the moment (October 2008). So Kermeta contracts are the current way you can put any kind of constraints on yout metamodel.

The ultimate goal of the Kermeta developers team is to be able to require an OCL file directly into Kermeta programs.

*college_constraints.kmt

```
@mainClass "CollegeMM::Verification"
@mainOperation "main"

package CollegeMM;

require kermeta
require "platform:/resource/org.openembedd.formations.samples.kermeta02/metamodel/CollegeMM.ecore"
require "college_lesson_rules.ocl"

class Verification
{
    operation main() : Void is do
        var rep : kermeta::persistence::Repository init kermeta::persistence::EMFRepository.new
        var res : kermeta::persistence::Resource init rep.getResource(
            "platform:/resource/org.openembedd.formations.samples.kermeta02/model/MyCollege.xmi")
//          "platform:/resource/org.openembedd.formations.samples.kermeta02/model/NotAGoodCollege.xmi")
        var model : CollegeMM::College
        model ?= res.one
        model.checkAllInvariants
    end
}
```

Chapter Two

Transforming models

II – Transformation: goal

- **An instance of a metamodel as input**
  - For example: a UML2 state machine
- **Another metamodel as target**
  - The Kermeta FSM corresponding instance
- **An automated process to produce the second**
  - First model as input
  - Our Kermeta code
  - Second model as output

Many other transformations than One-to-One can be realized, due to Kermeta ability to *compute* models

Transformations may occur on the input model itself, for adding features to it (like GOF patterns).

We may also:

- extract parts of a model,
- split it in multiple dedicated submodels,
- produce metrics about a set of models (like the number of classes, frequency of containment associations,...),
- ...

Models are the main data of MDE. You may consider a "data mining" approach, taking whole benefits of the information volume they represent.

II – Transformation : solution 1 (Visitor)

The *Visitor* pattern gives us the ability to navigate the input model and execute action(s) on each of its elements

- An `accept(Visitor)` method on existing classes
- Any number of concrete visitors, for tasks to be done
- A `visit(X)` method in the visitor for each metamodel element accepting X

Create references implies the relied instances must have been created

- First pass : instantiate objects (first concrete visitor)
- Second pass : rely them (second concrete visitor)

The output instances build leads to a *Builder* pattern

The original Visitor pattern was designed on languages which imply to add features to a class in the existing code.

With the Kermeta aspects, you can seamlessly weave the `visit()` code within the native classes.

As we can add behaviour to a metamodel without modifying it, we do not need the Visitor pattern, so we will present in the next slides an "aspect" way of writing transformations with Kermeta.

II – Transformation : solution 2 (aspects)

Kermeta "aspects" give us the ability to extend the metamodel behaviour as we need

- Add on each element
  - A first pass to create output element
  - A second pass to link output elements together
  - A browsing code to apply passes on sub-elements

The tree of UML2 input model

- uml::Model
  - uml::Package
    - uml::StateMachine
      - uml::Region
        - uml::Vertex
        - uml::Transition

II – Transformation : solution 2 (aspects)

Browser: navigate through input model

```
package uml;

require kermeta
require "http://www.eclipse.org/uml2/2.1.0/UML"

aspect class Element {
operation eachOwnedElement(func : <Element -> Element>)
                : Void is abstract
}

aspect class Model {
   method eachOwnedElement(func : <Element -> Element>)
                : Void is do
      self.packagedElement.each { o |
         func(o)
      }
   end
}

aspect class Package {
   method eachOwnedElement(func : <Element -> Element>)
                : Void is do
      self.packagedElement.each { o |
         func(o)
      }
   end
}
```

```
aspect class StateMachine {
   method eachOwnedElement(func : <Element -> Element>)
                : Void is do
      self.region.each{ o |
         func(o)
      }
   end
}

aspect class Region {
   method eachOwnedElement(func : <Element -> Element>)
                : Void is do
      self.subvertex.each{ o |
         func(o)
      }
      self.transition.each{ o |
         func(o)
      }
   end
}
```


As the browser is written for generic use, we will use it into our passes so we present it before passes code.

II – Transformation : solution 2 (aspects)

First pass: build target entities

```
package uml;

require kermeta
require "UmlBrowser.kmt"
require "http://www.kermeta.org/fsm"

// the main class in UML2 (all other classes derived from it)
aspect class Element {
   operation uml2fsmPass1() is abstract
}

aspect class Model {
   method uml2fsmPass1() is do
      self.eachOwnedElement{ p | p.uml2fsmPass1() } // browse
   end
}

aspect class Package {
   method uml2fsmPass1() is do
      self.eachOwnedElement{ p | p.uml2fsmPass1() } // browse
   end
}

aspect class StateMachine {
   reference output : fsm::FSM
   method uml2fsmPass1() is do
      output := fsm::FSM.new
      self.eachOwnedElement{ p | p.uml2fsmPass1() } // browse
   end
}
```

```
aspect class Region {
   reference outModel : fsm::FSM
   method uml2fsmPass1() is do
      self.eachOwnedElement{ p | p.uml2fsmPass1() }
// browse
   end
}

aspect class Vertex {
   reference output : fsm::State
   reference outModel : fsm::FSM
   method uml2fsmPass1() is do
      output := fsm::State.new
      output.name := self.name
   end
}

aspect class Transition {
                  // nothing to do as in FSM transitions
are links
}
```

II – Transformation : solution 2 (aspects)

Second pass: link target entities together

```
package uml;

require kermeta
require "UmlBrowser.kmt"
require "http://www.kermeta.org/fsm"

// the main class in UML2 (all other classes derived from it)
aspect class Element {
   operation uml2fsmPass2() is abstract
}

aspect class Model {
   method uml2fsmPass2() is do
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}

aspect class Package {
   method uml2fsmPass2() is do
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}

aspect class StateMachine {
   reference output : fsm::FSM
   method uml2fsmPass2() is do
      /* the region does not know directly its state machine
         so we must pass it to the Pass2 method */
      self.region.each{ r | r.outModel := self.output }
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}
```

```
aspect class Region {
   reference outModel : fsm::FSM
   method uml2fsmPass2() is do
      self.subvertex.each{ sv |
         sv.outModel := outModel
         outModel.ownedState.add(sv.output)
      }
      // browse
      self.eachOwnedElement{ p | p.uml2fsmPass2() }
   end
}

aspect class Vertex {
   reference output : fsm::State
   reference outModel : fsm::FSM
   method uml2fsmPass2() is do
      outModel.ownedState.add(output)
      var pseudoState : uml::Pseudostate
      pseudoState ?= self
      if pseudoState != void then
         if pseudoState.kind ==
               uml::PseudostateKind.initial then
            outModel.initialState := output
         end
      end
      var finalState : uml::FinalState
      finalState ?= self
      if finalState != void then
         outModel.finalState.add(output)
      end
   end
}

aspect class Transition {
   // nothing to do as in FSM transitions are links
}
```

II – Transformation: 2 (aspects)

Automated generation of the browser

- Copy the metamodel (xx.ecore) in local workspace
- Get the Kermeta MDK for Ecore
  - Fetch *fr.irisa.triskell.kermeta.ecore* from Kermeta CVS
  - In *src/kermeta/transformations*
    - Launch *ContainmentBasedActionPerformerGenerator.kmt*
      - With the local metamodel as parameter

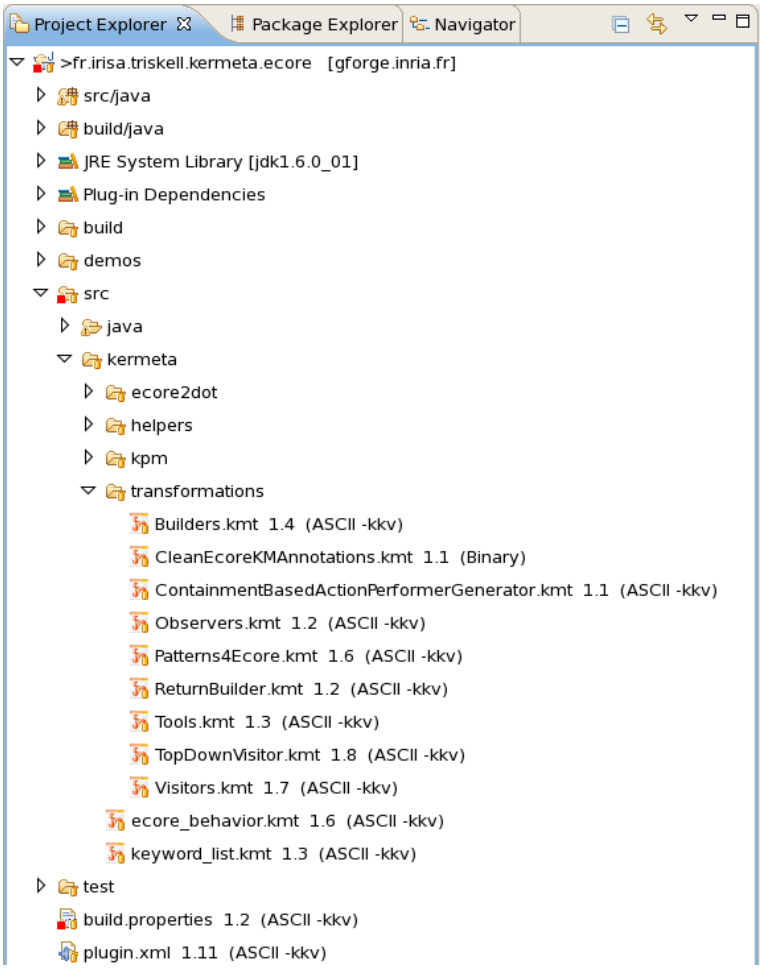
Obtain the browsing code

- In local workspace
  - *xx.ecore_ContainmentBasedActionPerformer.kmt*
- Rename it
  - *xx_browser.kmt*

You can now require it in your tasks code

Kermeta CVS:

- Host: gforge.inria.fr
- Repository: /cvsroot/kermeta
- Module: ecore_projects/
- Project: fr.irisa.triskell.kermeta.ecore



Simulating models

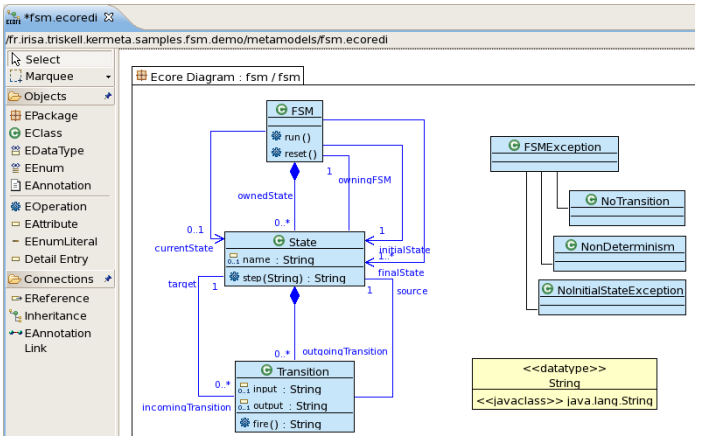
III – Simulation : principles

- **Execute a model means**
  - add behaviour to the metamodel
  - input data from the *Actor* who interacts with model
  - display the current state of the model under simulation
  - run the whole thing in an *Actor* (user) interface
- **Kermeta feeds all theses needs**
  - Aspects can define the code of operations
  - The Kermeta console can read strings
  - The Kermeta console can display strings
  - The Kermeta launcher can easily run *.km* or *.kmt* files

III – Simulation : the FSM example

- **The FSM metamodel**

  (see FSM tutorials inside Kermeta distribution)



The Finite State Machine metamodel is a state machine (like UML one) simple enough to permit interesting and light tutorials. The machine behavior is easily understandable and programable.

Kermeta tutorials which are based on the FSM metamodel:

- fr.irisa.triskell.kermeta.samples.fsm.demo
- fr.irisa.triskell.kermeta.samples.fsm.demoAspect
- org.kermeta.tutorial.aspects
- *How to add behavior to a metamodel*
- *How to run a Kermeta program*

III – Simulation : the FSM example

- **The FSM has three behaviour points**
  - **State** class `step(String) : String` method
    - Get the external input
    - Select another state depending on the input value
    - Return the output related to the involved transition
  - **Transition** class `fire() : String` method
    - Realize the action corresponding to the transition
    - Change the current state of the machine
    - Return the produced String
  - **FSM** class `run()` method
    - Plug the model instance into a simulator
    - Initialize the corresponding machine
    - Launch the whole execution process

The input can be a file containing some strings to parse, like a compiler parser, or characters put by a physical user on the simulator console.

The FSM run() method is not mandatory; the initialization and launch could be done by the simulator.

Despite this FSM.run() we need a Simulator class and a main() operation in order to load the model in memory before we can access to run() operation of its effective instance and execute it.

III – Simulation : the FSM example

- **Code part 1**

```
@mainClass "fsm::Simulator"
@mainOperation "main"

package fsm;

require kermeta
require "http://www.kermeta.org/fsm"

using kermeta::standard

aspect class State {
   operation step(c : String) : String raises FSMException is do
      var validTransitions : Collection<Transition>
      validTransitions := outgoingTransition.select{ t | t.input.equals(c) }

      if validTransitions.empty then raise NoTransition.new end
      if validTransitions.size > 1 then raise NonDeterminism.new end

      result := validTransitions.one.fire
   end
}

aspect class Transition {
   operation fire() : String raises FSMException is do
      source.owningFSM.currentState := target
      result := output
      stdio.writeln(" --> "+result)
   end
}
```


The return value (***result***) of an operation can be managed everywhere inside its body.

III – Simulation : the FSM example

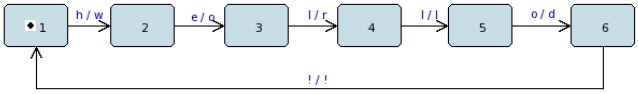
- **Code part 2**

```
aspect class FSM {
   operation run() : Void raises FSMException is do
      from var str : String init "init"
      until str == "quit"
      loop
         if( str != "init") then
            do
               var res : String
               res := self.currentState.step(str)
            rescue (err : NoTransition)
               stdio.writeln(err.toString)
            rescue (err : NonDeterminism)
               stdio.writeln(err.toString)
            end
         end
         stdio.writeln("Current state: " + self.currentState.name)
         str := stdio.read(" give me a string > ")
      end
   end
}

class Simulator {
   operation main(modelFile : String) : Void is do
      var rep : EMFRepository init EMFRepository.new
      var res : Resource init rep.getResource(modelFile)
      var model : FSM
      model ?= res.one
      model.currentState := model.initialState
      model.run
      stdio.writeln(" -----> exiting!")
   end
}
```

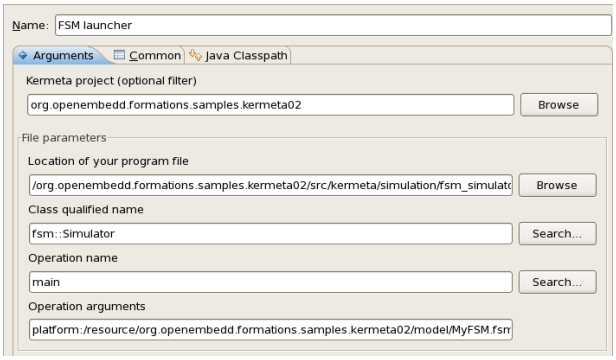

The simulator takes the full path model name as a parameter for Kermeta execution.

III – Simulation : the FSM example

- **A model to execute**



- **The execution parameters**



III – Simulation : the FSM example

- **The trace of an execution**