Meta-modeling with OCL & KerMeta

Prof. Jean-Marc Jézéquel
(Univ. Rennes 1 & INRIA)
Triskell Team @ IRISA
Campus de Beaulieu
F-35042 Rennes Cedex
Tel : +33 299 847 192 Fax : +33 299 847 171
e-mail : jezequel@irisa.fr
http://www.irisa.fr/prive/jezequel

The World and the Model

- A Model is a simplified representation of an aspect of the World for a specific purpose
  - Consider modeling both the machine & its environment (M. Jackson)
- UML paved the way from OOP to Model Based SE

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

\[ M_1 \] (modeling space)

\[ M_0 \] (the world)

© J.-M. Jézéquel, 2005
Assigning Meaning to Models

- If a UML model *is no longer* just
  - fancy pictures to decorate your room
  - a graphical syntax for C++/Java/C#/Eiffel...
- Then tools must be able to manipulate models
  - Let's make a model of what a model is!
  - $\Rightarrow$ *meta-modeling*
    » & meta-meta-modeling...
    » Use Meta-Object Facility (MOF) to avoid infinite Meta-recursion

---

The 4 layers in practice

© J.-M. Jézéquel, 2005
UML2 meta-model (part.)

Zoom: comments

This class was added by Alan Wright after meeting with the mission planning team.
Generalizations

Static Semantics with OCL

- Complementing a meta-model with Well-Formedness Rules, aka *Contracts* e.g.;
  - ModelElement has a unique name in a Namespace
  - no cycle in a UML2 inheritance graph...
- Expressed with the OCL (Object Constraint Language)
  - The OCL is a language of typed expressions.
  - A constraint is a valid OCL expression of type Boolean.
  - A constraint is a restriction on one or more values of (part of) an object-oriented model or system.
OCL

- Can be used at both
  - M1 level (constraints on Models)
    » aka Design-by-Contract (Meyer)
  - M2 level (constraints on Meta-Models)
    » aka Static semantics

- Let’s overview it with M1 level examples

Class invariants

- Contraints can be added to models
  - notation: between { }

- Invariant = Boolean expression
  - True for all instances of a class in stable states...
  - Expressed with the OCL (Object Constraint Language)
    » e.g. \{balance >= lowest\}
    » Can also navigate the associations

Bank_Account
{balance >= lowest}

balance: Money
lowest: Money

deposit (Money)
withdraw(Money)
Precondition:

**Burden on the client**

- Specification on what must be true for a client to be allowed to call a method
  - example: amount > 0

- Notation in UML
  - `{«precondition» OCL boolean expression}`
  - Abbreviation: `{pre: OCL boolean expression}`

Postcondition:

**Burden on the implementor**

- Specification on what must be true at completion of any successful call to a method
  - example: balance = balance @pre + amount

- Notation in UML
  - `{«postcondition» OCL boolean expression}`
  - Abbreviation: `{post: OCL boolean expression}`
  - Operator for previous value (idem old Eiffel):
    > OCL expression @pre
To be Abstract and Precise

```
Bank_Account
(balance>=lowest)

balance: Money
lowest: Money

deposit (amount: Money)
{pre: amount> 0}
{post: balance = balance @pre + amount}
withdraw(amount: Money)
{pre: amount> 0 and montant<=balance-lowest}
{post: balance = balance @pre - amount}
```

- In memory implementation
  - straightforward
  - list of transactions
- Data base implementation
  - etc.

Types in OCL

The types in OCL are as follows:

- Predefined types
  - Basic types - Integer, Real, String and Boolean
  - Collection types - Collection, Set, Bag, Sequence
- Meta types
  - OclAny, OclExpression, OclType
- User-defined model types
  - Enumeration and all classes, types and interfaces
### Boolean

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>a or b</td>
<td>Boolean</td>
</tr>
<tr>
<td>and</td>
<td>a and b</td>
<td>Boolean</td>
</tr>
<tr>
<td>exclusive or</td>
<td>a xor b</td>
<td>Boolean</td>
</tr>
<tr>
<td>negation</td>
<td>not a</td>
<td>Boolean</td>
</tr>
<tr>
<td>equals</td>
<td>a = b</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>a &lt;&gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>implication</td>
<td>a implies b</td>
<td>Boolean</td>
</tr>
<tr>
<td>if then else</td>
<td>if a then b1 else b2 endif</td>
<td>type of b</td>
</tr>
</tbody>
</table>

### Real and Integer

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>a = b</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>a &lt;&gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>less</td>
<td>a &lt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>more</td>
<td>a &gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>less or equal</td>
<td>a &lt;= b</td>
<td>Boolean</td>
</tr>
<tr>
<td>more or equal</td>
<td>a &gt;= b</td>
<td>Boolean</td>
</tr>
<tr>
<td>plus</td>
<td>a + b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>minus</td>
<td>a - b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>multiply</td>
<td>a * b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>divide</td>
<td>a / b</td>
<td>Real</td>
</tr>
<tr>
<td>modulus</td>
<td>a.mod(b)</td>
<td>Integer</td>
</tr>
<tr>
<td>integer division</td>
<td>a.div(b)</td>
<td>Integer</td>
</tr>
<tr>
<td>absolute value</td>
<td>a.abs</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>maximum</td>
<td>a.max(b)</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>minimum</td>
<td>a.min(b)</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>round</td>
<td>a.round</td>
<td>Integer</td>
</tr>
<tr>
<td>floor</td>
<td>a.floor</td>
<td>Integer</td>
</tr>
</tbody>
</table>
String

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>s.concat(string)</td>
<td>String</td>
</tr>
<tr>
<td>size</td>
<td>s.size</td>
<td>Integer</td>
</tr>
<tr>
<td>to lower case</td>
<td>s.toLower</td>
<td>String</td>
</tr>
<tr>
<td>to upper case</td>
<td>s.toUpper</td>
<td>String</td>
</tr>
<tr>
<td>substring</td>
<td>s.substring(int, int)</td>
<td>String</td>
</tr>
<tr>
<td>equals</td>
<td>s1 = s2</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>s1 &lt;&gt; s2</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Simple constraints

Customer

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td></td>
</tr>
<tr>
<td>title: String</td>
<td></td>
</tr>
<tr>
<td>age: Integer</td>
<td></td>
</tr>
<tr>
<td>isMale: Boolean</td>
<td></td>
</tr>
</tbody>
</table>

title = if isMale then ‘Mr.’ else ‘Ms.’ endif
age >= 18 and age < 66
name.size < 100
Using Enumerations

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>gender: enum(male, female)</td>
</tr>
<tr>
<td>name: String</td>
</tr>
<tr>
<td>title: String</td>
</tr>
<tr>
<td>dateOfBirth: Date</td>
</tr>
</tbody>
</table>

gender = #male implies title = ‘Mr. ‘

Non-local contracts: navigating associations

- Each association is a navigation path
  - The context of an OCL expression is the starting point
  - Role names are used to select which association is to be traversed (or target class name if only one)

Person 1 owner ownership ownings * Car

Context Car inv:
self.owner.age >= 18
Navigation of 0..* associations

Through navigation, we no longer get a scalar but a collection of objects

OCL defines 3 sub-types of collection

- **Set**: when navigation of a 0..* association
  - Context Person inv: ownings return a Set[Car]
  - Each element is in the Set at most once

- **Bag**: if more than one navigation step
  - An element can be present more than once in the Bag

- **Sequence**: navigation of an association {ordered}
  - It is an ordered Bag

Many predefined operations on type collection

Constraint examples

```plaintext
self.transaction -> forall(t:Transaction | t.value > 100)
```

![Diagram of Account and Transaction relationships]

- `self.balance > 0`
Navigating to collections

Customer

self.account

produces a set of Accounts

Customer

self.account.transaction

produces a bag of transactions

If we want to use this as a set we have to do the following

self.account.transaction -> asSet

Collection hierarchy

Collection

Set

- minus
- symmetricDifference
- asSequence
- asBag

Bag

- asSequence
- asSet

Sequence

- first
- last
- at(int)
- append
- prepend
- asBag
- asSet
Basic operations on collections

- **isEmpty**
  - true if collection has no element

- **notEmpty**
  - true if collection has at least one element

- **size**
  - Number of elements in the collection

- **count (elem)**
  - Number of occurrences of element elem in the collection

Context Person inv:
age<18 implies ownings->isEmpty

Multiplicity constraints

Vocabulary

VocabElement

Hint

Equivalent constraints expressed on the classes

VocabElement
self.hint -> size >= 0 and self.hint -> size <= 5

VocabElement
self.vocabulary -> size = 1

Hint
self.vocabElement -> size = 1
The subset constraint

Flight
self.crew \rightarrow \text{includes}( \text{self.pilot} )

Flight
self.crew \rightarrow \text{includesAll}(\text{self.flightAttendants})

select Operation

- possible syntax
  - collection->select(elem:T | expr)
  - collection->select(elem | expr)
  - collection->select(expr)

- Selects the subset of \textit{collection} for which property \textit{expr} holds

- e.g.
  - context Person inv:
    ownings->select(v: \text{Car} | \text{v.mileage}<100000)->\text{notEmpty}

- shortcut:
  - context Person inv:
    ownings->select(mileage<100000)->\text{notEmpty}
forall Operation

- possible syntax
  - collection->forall(elem:T | expr)
  - collection->forall(elem | expr)
  - collection->forall(expr)

- True iff expr holds for each element of the collection

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

- shortcut:
  ```
  context Person inv:
  ownerships->forall(mileage<100000)
  ```

Operations on Collections

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>The number of elements in the collection</td>
</tr>
<tr>
<td>count(object)</td>
<td>The number of occurrences of object in the collection.</td>
</tr>
<tr>
<td>includes(object)</td>
<td>True if the object is an element of the collection.</td>
</tr>
<tr>
<td>includesAll(collection)</td>
<td>True if all elements of the parameter collection are present in the current collection.</td>
</tr>
<tr>
<td>isEmpty</td>
<td>True if the collection contains no elements.</td>
</tr>
<tr>
<td>notEmpty</td>
<td>True if the collection contains one or more elements.</td>
</tr>
<tr>
<td>iterate(expression)</td>
<td>Expression is evaluated for every element in the collection.</td>
</tr>
<tr>
<td>sum(collection)</td>
<td>The addition of all elements in the collection.</td>
</tr>
<tr>
<td>exists(expression)</td>
<td>True if expression is true for at least one element in the collection.</td>
</tr>
<tr>
<td>forall(expression)</td>
<td>True if expression is true for all elements.</td>
</tr>
</tbody>
</table>
OCL for M2: Examples of WFR

- ModelElement has a unique name in a Namespace
  
  Context ModelElement inv:
  namespace.ownedElement->collect(name)->count(self.name)=1

- ...
Example

- A model
  ![State Transition Diagram](image)

- Its metamodel
  ![Metamodel Diagram](image)

- Adding Operational Semantics to OO Metamodels

From Metamodels to Languages

- Generic Syntax (HUTN)
- MOF EMOF ECore Infra Etc.
- Abstract Syntax
- Concrete Syntax
- Semantics

How do these things relate?

The language I want to build
Metadata languages

- (E)MOF => Only data structures
  - classes, properties, associations, ...
  - operations: only signatures

- Not sufficient to operate on models
  - Constraints
  - Actions
  - Transformations
  - ...

Typical example (excerpted from MOF spec)

**Operation isInstance(element : Element) : Boolean**

- "Returns true if the element is an instance of this type or a subclass of this type. Returns false if the element is null".

```plaintext
operation isInstance (element : Element) : Boolean is do
  // false if the element is null
  if element == void then result := false
  else
    // true if the element is an instance of this type
    // or a subclass of this type
    result := element.getMetaClass == self or
    element.getMetaClass.allSuperClasses.contains(self)
  end
end
```

A natural language specification

An operational specification
What is “meta”-executability?

- Basic CRUD Operations
- Merge, Composition...

► Simply an (object-oriented) program that manipulates model elements

“Program = Data Structure + Algorithm”, Niklaus Wirth

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.
Kermeta, a Kernel to Meta

EMOF $\iff$ Kermeta

```
class FSM {
    attribute ownedState : State[0..*]#owningFSM
    reference initialState : State[1..1]
    reference currentState : State
    operation run(c : String) : 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
    reference outgoingTransition : Transition[0..*]#source
    reference incomingTransition : Transition#target
    operation step(c : String) : kermeta::standard::String 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

\[
\begin{align*}
& a_1 : A \\
& b_1 : B \\
& a_2 : A \\
& b_1 := b_1 \\
& a_2.b := b_1 \\
\end{align*}
\]

After

**Association**

Before

\[
\begin{align*}
& c_1 : C \\
& d_1 : D \\
& c_2 : C \\
& d_2 := c_1 \\
& c_2 := c_1 \\
\end{align*}
\]

After

Example

**FSM**

- **State**
  - initialState
  - currentState
  - name: EString
  - state():
  - run():
  - reset():

- **Transition**
  - source
  - outgoingTransition
  - target
  - incomingTransition
  - name():

- **Operation**
  - fire() : String

- **Result**
  - 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 NonDeterminism.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.to_string)
        end
    end
end
stdio.writeln("** END OF SIMULATION **")
Kermeta workbench snapshot
Using aspect-composition to reflectively build Kermeta

The action metamodel

Close to the OCL

- CRUD operation
- Control structures
- Operation call
- Variables and assignment
- Exceptions handling
- Functions (OCL-like iterators)
Current Status

- Latest version (1.1.0)
  - Parser, type checker, interpreter, debugger
  - Eclipse plug-in: Textual Editor, Browser, Launcher
  - EMF Ecore metamodel Import / Export
  - EMF model Import / Export
  - Constraints (Kermeta or OCL)
  - Graphical Editor (generated with Topcased)
  - Documentation and Examples

- Under development / test
  - Seamless import of Java classes in Kermeta
  - Compiler
Smoothly interoperates with Eclipse/EMF
Open Source
► Download it now!

A statically typed object-oriented executable meta-language

- Home page
  - http://www.kermeta.org
- Development page
  - http://kermeta.gforge.inria.fr/