Safely Reusing Model Transformations through Family Polymorphism

Prof. Jean-Marc Jézéquel
Director of IRISA
jezequel@irisa.fr

Complex Software Intensive Systems

- Multiple concerns
- Multiple viewpoints
- Multiple domains of expertise
- => Needs to express them!
  - In a meaningful way for experts
    - Not everybody reads C code fluently…
Domain Specific Languages are Everywhere

- Why? Because *One size does not fit all!*

  ![Diagram showing the DSL Approach]

- Even variants of the same DSL co-exist
  - 50+ variants of StateCharts have been reported!

  ![Diagram showing the DSL Approach]

Typical lifecycle of a DSL

- ** Starts as a simple ‘configuration’ mecanism**
  - for a complex framework
- ** Grows more and more complex over time**
  - `ffmpeg -i input.avi -b:v 64k -bufsize 64k output.avi`
    - Cf [https://www.ffmpeg.org/ffmpeg.html](https://www.ffmpeg.org/ffmpeg.html)
- ** Evolves into a more complex language**
  - `ffmpeg config file`
    - A preset file contains a sequence of option=value pairs, one for each line, specifying a sequence of options. Lines starting with the hash (#) character are ignored and are used to provide comments.
  - Add macros, if, loops,…
    - might end up into a Turing-complete language!
DSL: From Craft to Engineering

- From supporting a single DSL…
  - Concrete syntax, abstract syntax, semantics, pragmatics
    - Editors, Parsers, Simulators, Compilers…
    - But also: Checkers, Refactoring tools, Converters…
- …To supporting Multiple DSLs
  - Interacting altogether
  - Each DSL with several flavors
  - And evolving over time
- Product Lines of DSLs!

Issues

- Shape of the DSL
  - Implicit = plain-old API to more fluent APIs
  - Internal or embedded DSLs written inside an existing host language (e.g. Scala)
  - External DSLs with their own syntax and domain-specific tooling.
- Language integration (cf. Gemoc)
- Support variants and evolution of DSLs
  - Backward compatibility, Migration of artifacts
  - Safe reuse of the tool chains
Focus of this talk

- Ease the definition of tool-supported DSL families
  - How to ease and validate the definition of new DSLs/tools?
  - How to correctly reuse existing tools?

⇒ From MDE to SLE… with **Model Typing**
  ⇒ static typing with models as first class entities
  - Focus: **reuse of model transformation** between several DSLs
Type Systems

- Type systems provide unified frameworks enabling many facilities:
  - Abstraction
  - Reuse and safety
  - Impact analyses
  - Auto-completion
  - ...
- What about a model-oriented type system?

Background: the OMG Meta-Modeling Stack

A Model is a simplified representation of an aspect of the World for a specific purpose.
Background: Executable Meta-Modeling

// 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.xtend”
// Transformations

class FSM {
  public def void reset() {
    currentState = initialState
  }
}

class Minimizer {
  public def FSM minimize (source: FSM) {…}
}

Model Type – motivation

• Motivating example: model transformation [SoSyM'07]
takes as input a state machine and produces a lookup table showing the correspondence between the current state, an arriving event, and the resultant state
  ⇒ side-effect free

  When can we reuse such a transformation?
Model Type – Further Needs

- Another example: optimizing compilers
  
  GeCos: C compiler infrastructure using Model Driven Engineering and Java. It leverages the Eclipse Modeling Framework and uses Eclipse as an underlying infrastructure.
  
  ⇒ The source language grammar & the IRs become metamodels.
  ⇒ Some of these DSLs present a graph structure
  ⇒ *dead code elimination* and circuit trimming use almost same algorithms
  ⇒ need to specialize it!!

```
if (true) {
  ...

  modeltype CDFG {
    operation elim() is do
      reachability
    end
  }

  modeltype Datapath extends CDFG {
    operation elim() is do
      revert
      super()
    end
  }
```

Model Type – motivation

- Issue when considering a model as a set of objects:
  - addition of a property to a class is a common evolution seen in metamodels
  - property = pair of accessor/mutator methods

⇒ subtyping for classes requires invariance of property types!!!
⇒ Indeed: adding a property will cause a covariant property type redefinition somewhere in the metamodel.
Class Matching [Bruce et al., ENTCS 1999]

- Substitutability of type groups cannot be achieved through object subtyping

Model Type – motivation

- Some (other) differences for objects in MOF:
  - Multiplicities on properties
  - Properties can be combined to form associations: makes checking cyclical
  - Need to check whether properties are reflexive or not
  - Containment (or not) on properties
Model Type – initial implementation

- Bruce has defined the matching relation (<#) between two type groups as a function of the object types which they contain
- Generalizing his definition to the matching relation between model type:
  
  Model Type \( M' <# M \) iff for each object type \( C \) in \( M \) there is a corresponding object type with the same name in \( M' \) such that every property and operation in \( M.C \) also occurs in \( M'.C \) with exactly the same signature as in \( M.C \).

  \[ \text{matching} \cong \text{subtyping \ (by group)} \]

Application to MOF-Class Matching

- \( C_1 \) matches \( C_2 \) (\( C_1 <# C_2 \)) iff:
  - Same names
  - If \( C_1 \) is abstract, it can only match another abstract class
  - For each \( C_2 \) operation, \( C_1 \) must have a corresponding operation:
    - With the same name
    - With covariant return type
    - With corresponding parameters
    - In the same order
    - With contravariant types
    - With the same multiplicities
    - With the same isUnique attribute
  - For each \( C_2 \) property, \( C_1 \) must have a corresponding property:
    - With the same name
    - With covariant type
    - With the same multiplicities
    - With the same isUnique
    - With the same isComposite
    - With an opposite with the same name
  - Every mandatory property in \( C_1 \) must correspond to a \( C_2 \) property
Model Type – initial implementation

<table>
<thead>
<tr>
<th>Type</th>
<th>Simple (Figure 4)</th>
<th>Multiple-Start (Figure 5)</th>
<th>Mandatory-Start (Figure 6)</th>
<th>Composite (Figure 7)</th>
<th>With-Final-States (Figure 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I’ matches →</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Example code:

```java
class Serializer<MT : basic_fsm_type> {
    modeltype basic_fsm_type {
        basic_fsm :: FSM,
        basic_fsm :: State,
        basic_fsm :: Transition
    }

    modeltype finalstates_fsm_type {
        finalstates_fsm :: FSM,
        finalstates_fsm :: State,
        finalstates_fsm :: Transition,
        finalstates_fsm :: FinalState
    }

    Basic FSM Model Type          Final States FSM Model Type

A Basic FSM Operation Applied on a Final States FSM
```
Model Type – initial implementation

- **Supports:**
  - the addition of new classes (FinalState)
  - the tightening of multiplicity constraints (Mandatory)
  - the addition of new attributes (indirectly with Composite State Charts, via the added inheritance relationship)
  - Match-bounded polymorphism

- **Does not support:**
  - multiple initial states: accessing the `initialState` property in Basic state machine will return a single element typed by `State` while in Multiple state machine it will return a `Collection<State>`

  => technical nightmare!

---

Model Type – enhancing matching relation

- **Issues:**
  - metamodel elements (e.g., classes, methods, properties) may have different names.
  - types of elements may be different.
  - additional or missing elements in a metamodel compared to another.
  - opposites may be missing in relationships.
  - the way metamodel classes are linked together may be different from one metamodel to another
1  comment inférer si l'addition n'a pas d'impact ?
   Par exemple si l'ajout est obligatoire dans un objet
   instancié par la transformation.
   ==> exception !
   Benoit Combemale; 21/09/2011

2  ne peut-il pas être détecté et générer automatiquement les
   adapteur ?
   Benoit Combemale; 19/09/2011
Model Type – enhancing matching relation

• Motivating example: model refactoring [MODELS'09]

PULL UP METHOD: moving methods to the superclass when methods with identical signatures and results are located in sibling subclasses.

⇒ Model refining (with side-effect)

How to reuse such transformation?

Model Type – enhancing matching relation

Model Type $M'$ matches another model type $M$ (denoted $M' <# M$) iff for each class $C$ in $M$, there is one and only one corresponding class or subclass $C'$ in $M'$ such that every property $p$ and operation $op$ in $M.C$ matches in $M'.C'$ respectively with a property $p'$ and an operation $op'$ with parameters of the same type as in $M.C$.

• In practice to specify generic model refactorings:
  1. specify a lightweight metamodel (or model type) that contains the minimum required elements for refactorings.
  2. specify refactorings based on the lightweight metamodel.
  3. adapt the target metamodels using Kermeta for weaving aspects adding derived properties and opposites that match with those of the generic metamodel.
  4. apply the refactoring on the target metamodels
**Model Type – enhancing matching relation**

1. **Generic Model Type for the Pull Up Method Refactoring**

   ```plaintext
   aspect class Refactor-MT : GenericMT{
       operation pullUpMethod( source : MT; Class, target : MT; Class, meth : MT; Method ) : Void
       // Preconditions
       pre sameSignatureInOtherSubclasses is do
       target.subClasses,forall( s | s.
       sub.methods,exists( { op | haveSameSignature(op, meth) } ) )
       end
       // Operation body
       is do
       target.methods.add(meth)
       source.methods.remove(meth)
       end
   }
   ```

2. **Kermeta Code for the Pull Up Method Refactoring**

3. **Kermeta Code for Adapting the Java Metamodel**

   ```plaintext
   package java;
   require "Java.ecore"
   aspect class Classifier {
       reference inv_extends : Classifier[0..*]#extends
       reference extends : Classifier[0..1]#inv_extends
   }
   aspect class Class {
       property superClasses : Class[0..1]#subClasses
       getter is do
       result := self.extends
       end
       property subClasses : Class[0..*]#superClasses
       getter is do
       result := java::OrderedSet.new.
       self.inv_extends.each { subC | result.add(subC) }
       end
   }
   ```
Model Type – enhancing matching relation

4 Kermeta Code for Applying the Pull Up Method

```kermeta
package refactor;

require "http://www.eclipse.org/uml2/2.1.2/UML"

class Main {
  operation main() : Void is do
    var rep : EMFRepository init EMFRepository.new
    var model : uml::Model
      ?= rep.getResource("lan_application.uml").one
    var source : uml::Class init getClass("PrintServer")
    var target : uml::Class init getClass("Node")
    var meth : uml::Operation init getOperation("bill")
    var refactor : refactor::Refactor<uml::UnIMM>
      init refactor::Refactor<uml::UnIMM>.new
    refactor::pullUpMethod(source, target, meth)
}
```

Bottom Line: Model Subtyping Relations

- Are models typed by MT1 substitutable to models typed by MT2?

- Two criterions to be considered
  - Structural heterogeneities between the model types
  - Context in which the subtyping relation is used
Structural heterogeneities

- **Isomorphic**
  - MT1 possesses the same structure as MT2
  - Comparison using class matching
- **Non-isomorphic**
  - Same information can be represented under different forms
  - Model adaptation from MT1 to MT2

Context of use

- **Total**
  - We can safely use a model typed by MT1 everywhere a model typed by MT2 is expected
- **Partial**
  - We can safely use a model typed by MT1 in a given context where a model typed by MT2 is expected
  - I.e., reuse of a given model manipulation \( m \)
  - MT1 must possess all the information needed for \( m \)
  - I.e., the effective model type of \( m \) from MT2
4 Model Subtyping Relations

- Total isomorphic
  Matching
- Partial isomorphic
  + Pruning
- Total non-isomorphic
  + Adaptation
- Partial non-isomorphic
  + Pruning + Adaptation

Conclusion on Model Sub-Typing

- Current state in model typing
  - reuse of model transformations between isomorphic graphs
  - deal with structure deviation by weaving derived properties

⇒ *Statically checked in Kermeta!!*
Model Type – *Further Needs in a Model Type System*

- **Issues:**
  - New DSLs are not created from scratch
    ⇒ DSLs family (e.g., graph structure)
  - Model transformations cannot yet be specialized
    ⇒ call to `super` and polymorphism
  - Reuse through model type matching is limited by structural conformance
    ⇒ use of (metamodel) mapping
  - Chains of model transformations are fixed & hardcoded
    ⇒ partial order inference of model transformations

---

Wrap-up: Challenges

- **Reuse**
  - language constructs, grammars, editors or tool chains
    (model transformations, compilers…)

- **Substitutability**
  - replacement of one software artifact (e.g. code, object, module) with another one under certain conditions

- **Extension**
  - introduction of new constructs, abstractions, or tools
a voir pourquoi ?
Benoit Combemale; 19/09/2011
Challenges for DSL Modularity

➢ Modularity and composability
  ▪ structure software applications as sets of interconnected building blocks

➢ How to breakdown a language?
  ▪ how the language units should be defined so they can be reused in other contexts
    • What is the correct level of granularity?
    • What are the services a language unit should offer to be reusable?
    • What is the meaning of a service in the context of software languages?
    • What is the meaning of a services composition in the context of software languages?

Challenges for DSL Modularity

➢ How can language units be specified?
  • not only about implementing a subset of the language
  • but also about specifying its boundary
    – the set of services it offers to other language units
    and the set of services it requires from other language units.
  • classical idea of required and provided interfaces
    – introduced by components-based software engineering approaches.
    – But... What is the meaning of “provided and required services” in the context of software languages?
  • composability & substitutability
    – Extends vs. uses
Challenge: Variability Management and Languages Families

- Family of languages
  - Like in Software Product Line Engineering

- Alignment with the modularization approach
  - Need for a ‘unit’ that can, or cannot, be there

- Multi-stage orthogonal variability modeling
  - one language construct (i.e., a concept in the abstract syntax)
    - may be represented in several ways (i.e., several possible concrete syntaxes)
    - and/or may have different meanings (several possible semantics)

3 Dimensions of Variability

- Abstract syntax variability
  - functional variability
    - E.g. Support for super states in StateCharts

- Concrete syntax variability
  - representation variability
    - E.g. Textual/Graphical/Color…

- Semantics variability
  - interpretation variability
    - E.g. Inner vs outer transition priority
Big Picture: Variability Everywhere

- Variability in Metamodeling:
  - Semantic variation point
  - DSML Families
  - Knowledge capitalization
  - Language Engineering

- Variability in Modeling:
  - Support positive and negative variability
  - Derivation semantics must take into account the assets language semantics

Challenges: Verification & Validation

- Questions:
  - is a language really suited for the problems it tries to tackle?
  - Can all programs relevant for a specific domain be expressed in a precise and concise manner?
  - Are all valid programs correctly handled by the interpreter?
  - Does the compiler always generate valid code?
- => Design-by-Contract, Testing
Conclusion

- From supporting a single DSL…
  - Concrete syntax, abstract syntax, semantics, pragmatics
    - Editors, Parsers, Simulators, Compilers…
    - But also: Checkers, Refactoring tools, Converters…
- …To supporting Multiple DSLs
  - Interacting altogether
  - Each DSL with several flavors: families of DSLs
  - And evolving over time
- Product Lines of DSLs
  - Share and reuse assets: metamodels and transformations

Acknowledgement

- All these ideas have been developed with my colleagues of the DiverSE team at IRISA/Inria