Bidirectional Incremental Transformations with Active Operation Framework - Application to Facades

O. Beaudoux, F. Jouault, ESEO/TRAME

Context

- Synchronizing models is a significant unsolved issue in the MDE community.
  - Model synchronization is about maintaining a correspondence between a set of models.
- Façade project illustrates such a need by requiring the joint use of UML profiles and metamodels for specifying DSML, thus reaping benefits from both approaches.
- Façade was first built as an ad hoc solution of model synchronization.

Proposed Solution

- The Active Operation Framework (AOF) has been developed by ESEO Group to synchronize (meta)models.
- AOF defines incremental algorithms for OCL-like functions defined on top of OCL collections.
- They have already been implemented by three AOF API (in Scala, JavaScript, and Java).
- **The proposed solution consists in using AOF to synchronize models within the Façade project.**

Results

- The project is split into three packages that respectively consist in:
  1. **Extending the AOF Java API so that it can operate on EMF models for synchronizing them;**
  2. **Applying this implementation to the Façade project;**
  3. **Applying Façade to a UML-RT case study.**
- The packages will be distributed as Eclipse plug-ins under EPL, and committed to Papyrus.

Definition: Transformation

- A **transformation** *t* creates a set of target models *b* from a set of source models *a*.

  `b  = t(a )`

- If *a* later changes into *a'* = *c(a)*, it becomes desynchronized with b. One can compute a **new** target model *b'*.

  `b′ = t(a′) = t(c(a))`

- If *b'* later changes into *b''*, it becomes desynchronized from the source model.

Definition: Incremental Transformation

- An **incremental transformation** stays active after creating the initial target models, and propagates source model changes.

```
b  = t(a )
↓②      ↓①
b′ = t(a′) = t(c(a))
```

- It works by applying *c'* corresponding to *c*:

```
c′(b) = t(c(a))
```

- If *b'* later changes into *b''*, it becomes desynchronized from the source model.

Definition: Bidirectional Incremental Transformation

- A **bidirectional incremental transformation** stays active after creating the initial target models and propagates both source and target model changes. e.g.:

```
b              = t(a )
↓①                ↓②
b′ = c′(b) = t(a′) = t(c(a))
```

- It works by applying *c* corresponding to *c’*:

```
c′(b) = t(c(a))
```

Active Operations & Models

- Active operations are bidirectional incremental transformations on boxes (i.e., collections or wrapped singletons).
- A model is an object graph...
  - Edges are represented with object slots. This is how EMF works.
- … with every slot defined as a box.

Available Operations

- Operations on boxes are well-known functional operations notably found in OCL:
  - select, collect, size, isEmpty, notEmpty, union, etc.
- Plus a few specific operations:
  - collectTo, collectedFrom, bind, switchCollect
- Most operations are at least incremental and are bidirectional if provided with enough information.

Example: Derived Features (1 Of 2)

-- Specification: all resources must be used by transports.

-- Directly actionable using active operations.

context Transporter **inv**:

self.resources = self.transports.driver->union(

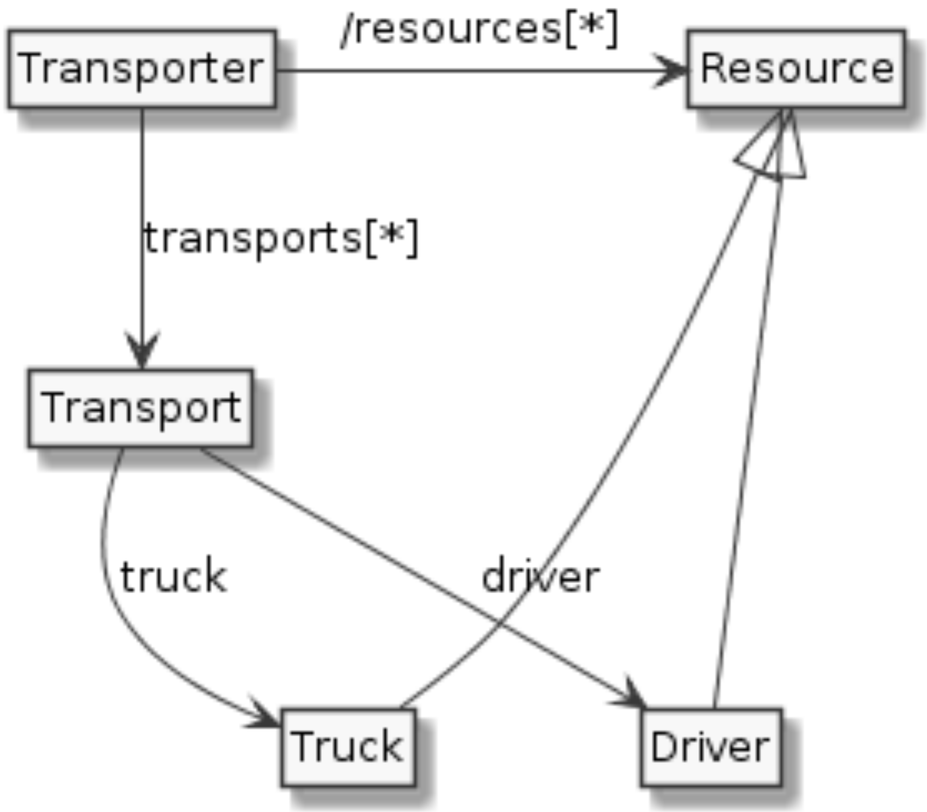
self.transports.truck)

-- Classical approach:

-- implementation of /resources derived feature.

context Transporter::resources : OrderedSet (Resource) **derive**:

self.transports.driver->union(self.transports.truck)



Example: Derived Features (2 Of 2)

-- Specification: resources are the union of drivers and

-- trucks. Directly actionable using active operations.

context Transporter **inv**:

self.resources = self.drivers->union(self.trucks)

-- Classical approach:

-- 1) implementation of /drivers derived feature

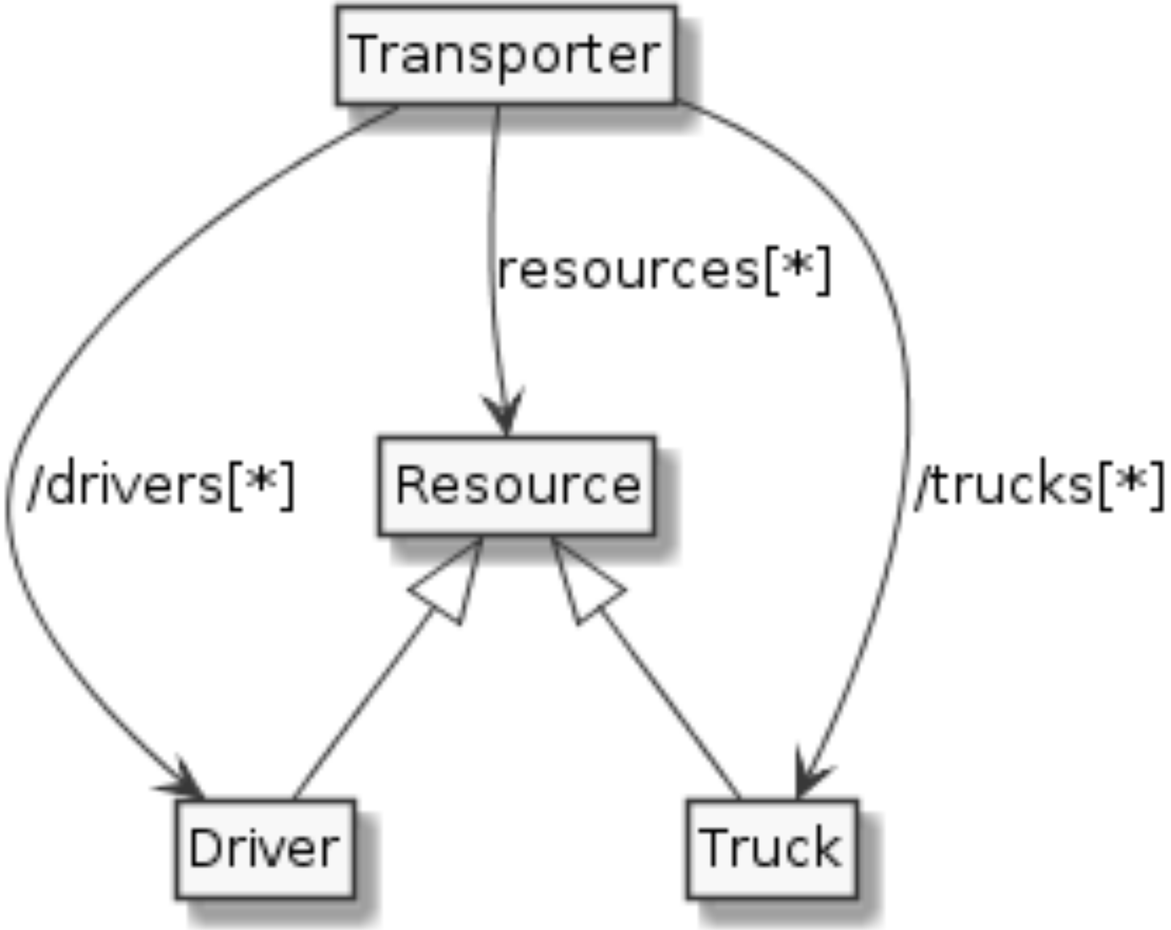
context Transporter::drivers : OrderedSet(Driver) **derive**:

self.resources->select(e | e.oclIsKindOf(Driver))

-- 2) implementation of /trucks derived feature

context Transporter::trucks : OrderedSet(Truck) **derive**:

self.resources->select(e | e.oclIsKindOf(Truck))



Box Types

	Box Type Name	Multiplicity	Unique	Ordered
No direct OCL equivalent	Option	0..1	N/A (true by convention)	
	One	1..1		
Same type name as in OCL	Set	0..*	true	false
	Bag		false	false
	OrderedSet		true	true
	Sequence		false	true

collect: unidirectional version

OCL example:

```
def: multBy2(set : Set(Integer)) : Set(Integer) =
        set->collect(e | 2*e)
```

AOF equivalent:

```
IBox<Integer> multBy2(ISet<Integer> set) {
        return set.collect(e -> 2*e);
}
```

Active evaluation:

- Adding *e* in *set* results in adding *2*e* in the target
- Removing *e* from *set* results in removing *2*e* from the target
- Replacing *p* by *e* in *set* results in replacing *2*p* by *2*e* in the target

Limitation to immutable states:

- Function *(e | 2*e)* operates on immutable states: *e* can never mutate, it can only be added, removed or replaced in box *set*
- Active evaluation of *collect* is limited to functions that do **not** manipulate mutable state of its parameter: these functions are called **immutable functions**

collect: bidirectional version

AOF bidirectionality:

```
IBox<Double> multBy2(ISet<Double> set) {
        return set .collect(e -> 2*e, e-> e/2);
}
```

Reverse function:

- The reverse function e-> e/2 only exists if the forward function is **bijective**
- Example: e->2*e is not bijective on integers

Active evaluation:

- Adding *e* in the target results in adding e/2 in box *set*
- Removing *e* from the target results in removing e/2 from box *set*
- Replacing: same principle

Infinite cycle:

Avoided by ignoring changes while propagating one

collectMutable

OCL example:

def: allChildrenOf(pop : Set(Person)) : Set(Person) =

```
    pop->collect(p | p.children)
```

AOF equivalent:

```
IBox<Person> allChildrenOf(ISet<Person> pop) {
    return pop.collectMutable(p -> p.getChildren());
}
```

Active evaluation:

- Adding *p* in *pop* results in
  - adding all elements of *p.children* in the target
  - observing *p.children*:
    - Adding *c* in *p.children* results in adding *c* in the target
    - Removing *c*, Replacing *c*: same principle
- Removing *p*, Replacing *p*: same principle

Bidirectionality:

- *collectMutable* is bidirectional but with some **restrictions**
- Box *pop* must contain a single *p*: it must be a **singleton** (an *Option* or *One* box)
- Adding child *c* in the target results in adding *c* in the children of *p*

switchCollect: Collecting Immutable Values Depending on Mutable Conditions.

```
def: switchCollectExample(integers:Set
(Integer)) : Set(Integer) =
    integers->collect(i |
        if i.mod(2) <> 0 then
            -i
        else if i.mod(3) = 0 then
            i*i
        else
            i
        endif endif
    )
```

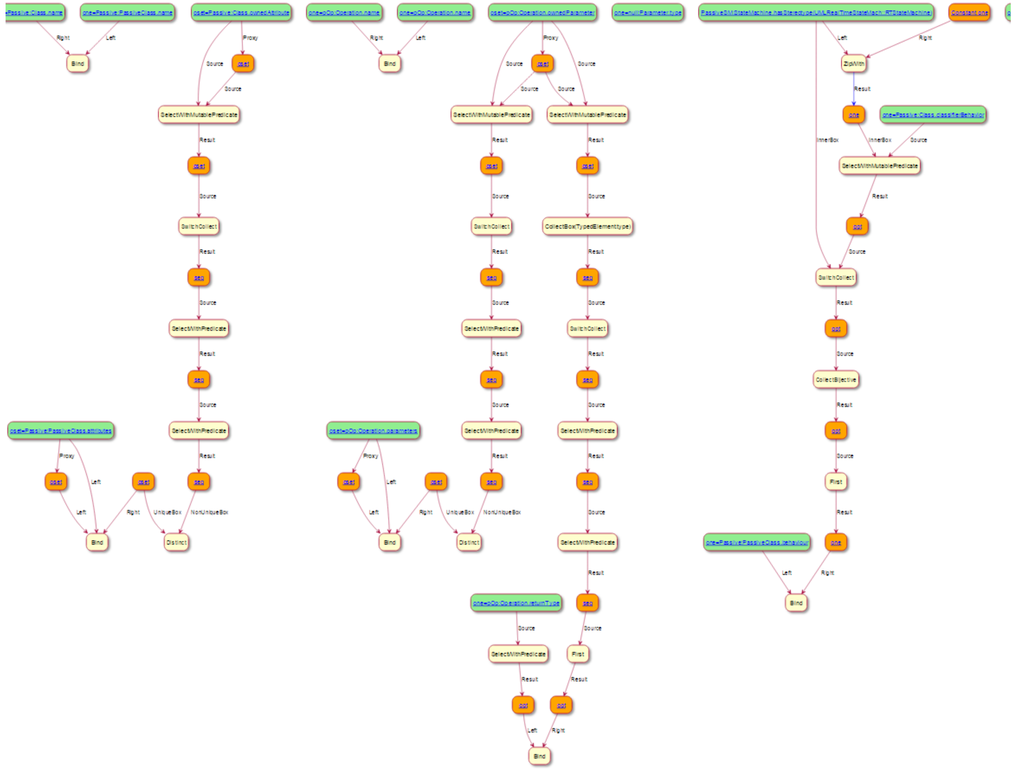
```
IBox<Integer> switchCollectExample(ISet<Integer> integers) {
  return integers.switchCollect(
    newArray(i -> (i % 2) != 0, i -> (i % 3) == 0),
    newArray(i -> -i, i -> i*i),
    i -> i
  )
}
```

Traceability: collectTo & collectedFrom

- *collectTo* computes a set of mutable objects by applying a collector function to each source element.
  - It uses a cache to always return the same target object for a given source object.
  - It requires bijectivity of the collector function.
- *collectedFrom* enables retrieving a source element from a target element.
- A reverse collector function can be given to *collectTo*.
  - New elements can then be inserted in its result.

Debugging

- *inspect* pseudo-operation
  - logs every change to the box on which it is applied
- Boxes interconnections
  - graph visualization
  - expressions serialization



Testing

- Testing passive aspect of an AOF-based transformation is like testing any transformation.
- Testing forward change propagation requires a specific approach:
  1. Execute transformation once: $b = t(a)$
  2. Perform some change on a: $c(a)$ resulting in a change on b: $c'(b)$
  3. Execute transformation again: $b' = t(c(a))$
  4. Compare $c'(b)$ and b'
- Testing reverse change propagation follows the same scheme except b is changed at step 2.

Façade Demo

- Javadoc overview
- Basic UI for interactive demo
- Pipes overview
- Debugging support: detailed formatters

Rule Encoding

```
rule Class2Capsule extends Class2Entity {
	from
		source : UML::Class (
			source.isStereotypeApplied(UMLRealTime::Capsule)
		)
	to
		target : UMLRT::Capsule (
			-- bindings here
		)
}
```

```
addRule(new Rule<Class, EObject>(this, "Class2Capsule", UML_Class, UMLRT_Capsule, UMLRealTime_Capsule) {
	public void bindContents(Bindings<Class, EObject> bindings, Class source, final EObject target) {
		bindEntityContents(bindings, source, target);

		// bindings here
	}
});
```

Binding Encoding

```
target.parts :=:    source.ownedAttribute->select(e |
                        e.isStereotypeApplied(UMLRealTime::CapsulePart)
                    )
```

```
bindings.add(target, "parts", source, "ownedAttribute",
    isStereotypeApplied(UMLRealTime_CapsulePart)
);
```

Abstract Rule Encoding

```
abstract rule NamedElement {
    from
        source : UML::NamedElement
    to
        target : UMLRT::NamedElement (
            target.name :=: source.name
        )
}
```

```
private <S extends NamedElement> void
    bindNamedElementContents(Bindings<S, EObject> bindings, S source,
                                        EObject target) {
    bindings.add("name", target, source);
}
```

Rule Guard Encoding

```
rule Class2Passive extends Class2Entity {
	from
		source : UML::Class (
			not (
				source.oclIsKindOf(UML::StateMachine) or
				source.isStereotypeApplied(UMLRealTime::Capsule)
			)
		)
	to
		target : UMLRT::PassiveClass
}
```

```
addRule(new Rule<Class, EObject>(this, "Class2Passive", UML_Class, UMLRT_PassiveClass,
	MutablePredicate.<Class>isInstanceOf(UML_StateMachine).or(
		isStereotypeApplied(UMLRealTime_Capsule)).not()
) {
	public void bindContents(Bindings<Class, EObject> bindings, Class source, EObject target) {
		bindEntityContents(bindings, source, target);
	}
});
```

Complex Select Example

```
target.attributes :=:     source.ownedAttribute->reject(e |
                                e.oclIsKindOf(UML::Port) or
                                e.isStereotypeApplied(UMLRealTime::CapsulePart)
                          ),
```

```
bindings.add(  target, "attributes",
               source, "ownedAttribute",
                   isInstanceOf(UML_Port)
                   .or(
                       isStereotypeApplied(UMLRealTime_CapsulePart)
                   ).not()
            );
```

Target Selection for Target Errors Detection

```
target.redefines->select(e |
                e.oclIsKindOf(UMLRT::Entity)
            ) :=: source.redefinedClassifier->reject(e |
                    e.oclIsKindOf(UML::Collaboration)
                )
```

```
bindings.add(
    target, "redefines",
            Predicate.isInstanceOf(UMLRT_Entity),
    source, "redefinedClassifier",
            isInstanceOf(UML_Collaboration).not()
);
```

Some Limitations

- Façades must be initially applied in the forward direction (i.e., on a profiled UML model).
- Maximum one rule per target type.
- Unloading a Façade currently requires unloading both source and target models.
- Some transformation structures are not supported yet in reverse. But we have prototype resolutions for several.
- Performance (speed & memory footprint) not studied yet.
  - But several simple optimizations already identified.

Some Perspectives

- Use OCL as a surface query & transformation language for active operations that allows:
  - writing mappings between models,
  - running these mappings by transforming them into an executable form based on AOF Java API.
- Apply AOF transformations to problems beyond Façade.
  - e.g., to the definition and execution of mappings between domain models and diagrams (based on Java API such as GEF/Draw2d, or JavaFX). See afternoon presentation.
- Overcome current limitations (e.g., predicate reversal) and optimize (e.g., lazy evaluation)

Model Synchronization Benchmark?

- Objective: enable easier comparison of model synchronization solutions
- Approach: define a benchmarking framework automatically measuring
  - propagation coverage (e.g., forward-only, reverse, complex cases)
  - performance (e.g., initial speed, propagation speed, memory footprint)
- Proposing it to TTC 2016?

Questions?