An Implementation of QVT Bidirectional Transformations

Executing QVT-R over UML+OCL Models using Alloy

Nuno Macedo Alcino Cunha

HASLab — High Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal

FATBIT/SSaaPP Workshop
September 17, 2012, Braga

Introduction

- Model-driven engineering involves multiple models representing possibly overlapping information;
- OMG has proposed standards for the specification of models (UML) and constraints over them (OCL);
- The *QVT* (Query/View/Transformation) language has been proposed to specify *bidirectional transformations* (BX).

Query/View/Transformation

- The QVT standard proposes three different specification languages;
- We focus on *QVT Relations* (QVT-R), a declarative language;
- Specifications define relations between elements of the model;
- Two running modes:
  - *checkonly* mode (checks consistency);
  - *enforce* mode (propagates updates in order to restore consistency): check-before-enforce (consistent models are not updated).

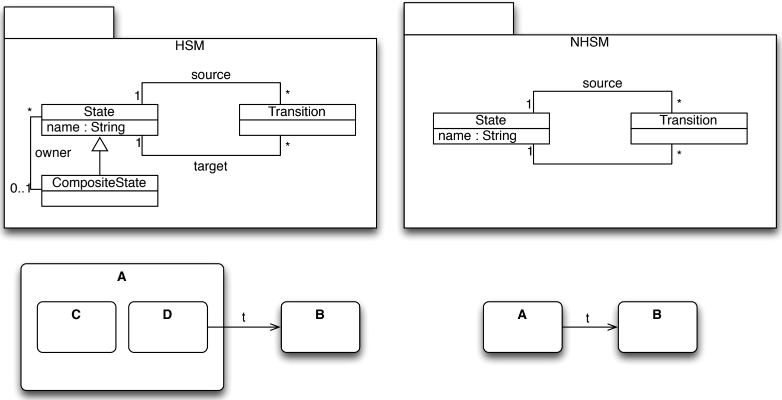
Bidirectional Transformations

- BXs are artifacts that represent the transformations in both directions;
- We need to infer both transformations from a single QVT-R specification;
- Since models may contain different information, they are *not bijective*;
- Propagating updates from a source to a new target retrieves information from the original target.

QVT Relations

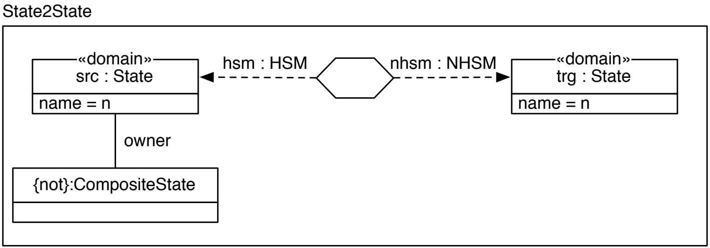
- A QVT-R *transformation* consists of set of QVT-R *relations* between elements of the models;
- In each relation there is a set of *domain patterns* that specify related elements;
- It may also contain *When* and *Where* clauses, that act as pre- and post-conditions.

Example: Expand/Collapse State Diagrams



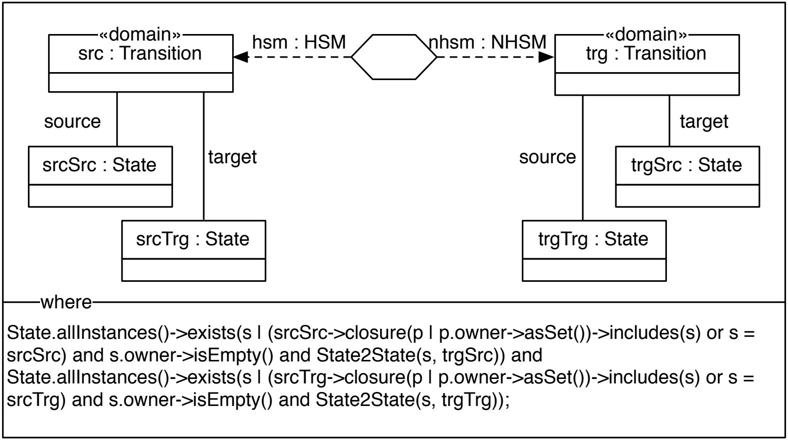
Example: *State2State*

```
top relation State2State {
    n : String;
    domain hsm src : State {
        name = n,
        owner = null
    };
    domain nhsm trg : State {
        name = n
    };
}
```



Example: *Transition2Transition*

Transition2Transition



Properties

- For every QVT transformation T between M and N we have:
  - a relation $\mathbf{T} : M \to N$ that checks the consistency;
  - transformations $\overrightarrow{\mathbf{T}} : M \times N \to N$ and $\overleftarrow{\mathbf{T}} : M \times N \to M$ that propagate updates;
- These artifacts are also inferred at the QVT relation level, between elements of the models;
- For every metamodel M, we have a function $\Delta_M : M \times M \to \mathbb{N}$ that calculates the distance between instances.

Properties

- Correctness:

$$\forall\, m \in M, n \in N : m\ \mathbf{T}\ (\overrightarrow{\mathbf{T}}\ (m, n))$$
$$\forall\, m \in M, n \in N : (\overleftarrow{\mathbf{T}}\ (m, n))\ \mathbf{T}\ n$$

- Hippocraticness (check-before-enforce):

$$\forall\, m \in M, n \in N : m\ \mathbf{T}\ n \Rightarrow m = \overrightarrow{\mathbf{T}}\ (m, n) \wedge n = \overleftarrow{\mathbf{T}}\ (m, n)$$

Properties

- Correctness:

$$\forall\, m \in M, n \in N : m\ \mathbf{T}\ (\overrightarrow{\mathbf{T}}\ (m, n))$$
$$\forall\, m \in M, n \in N : (\overleftarrow{\mathbf{T}}\ (m, n))\ \mathbf{T}\ n$$

- Hippocraticness (check-before-enforce):

$$\forall\, m \in M, n \in N : m\ \mathbf{T}\ n \Rightarrow m = \overrightarrow{\mathbf{T}}\ (m, n) \wedge n = \overleftarrow{\mathbf{T}}\ (m, n)$$

- Principle of least change ($\Rightarrow$ hippocraticness for $\Delta = 0$):

$$\forall\, m \in M, n, n' \in N : m\ \mathbf{T}\ n' \Rightarrow \Delta_N\ (\overrightarrow{\mathbf{T}}\ (m, n), n)\ \leqslant\ \Delta_N\ (n', n)$$
$$\forall\, m, m' \in M, n \in N : m'\ \mathbf{T}\ n \Rightarrow \Delta_M\ (\overleftarrow{\mathbf{T}}\ (m, n), m)\ \leqslant\ \Delta_M\ (m', m)$$

QVT-R Semantics

- The semantics of a QVT transformation consist of running its constituent QVT relations;
- *Check semantics*: for all candidate elements in the domain there must exist a candidate element in the target that matches it;
- *Enforce semantics*:
  - for all elements in the domain, if there is not a match in the target, update any element to match;
  - for all elements in the target, if it is not matched to an element in the domain, remove it.

QVT-R Semantics

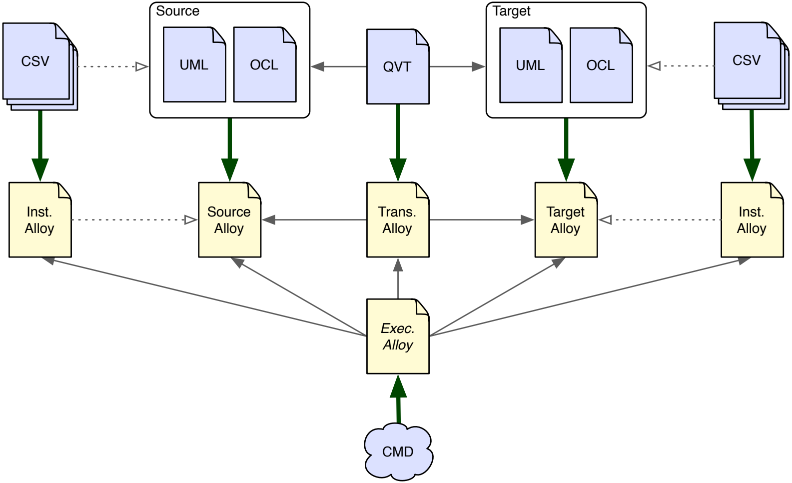
Adoption of QVT as a standard has been slow:

- The standard is ambiguous and incomplete regarding semantics;
- Tools implement different interpretations or disregard it at all;
- Some work on the formalization of the check semantics has been done...
- ...but not on the formalization enforce semantics;
- No tool has support for enforce mode over models with OCL constraints.

Alloy

- Alloy is a lightweight model-checking tool based on relational calculus;
- Allows automatic bounded verification of properties and generation of instances;
- We have already developed a tool for the transformation of UML+OCL class models to Alloy;
- Building up on that, we propose the translation of QVT-R to Alloy.

QVT to Alloy Translation



Models

- UML classes and their attributes are directly translated to Alloy signatures and relations;

```
sig State {
  state : set Instance,
  name : String -> Instance,
  owner : CompositeState -> Instance }
```

- Alloy is static, so we resort to the local state idiom;

```
fact {
  all i : Instance | name.i in (state.i -> one String) &&
    owner.i in (state.i -> lone compositestate.i) }
```

- OCL is also translated to constraints in Alloy.

Transformations: Check semantics

- We follow the check semantics of the standard;
- Each domain pattern produces a predicate in Alloy that represents candidate elements;

```
pred PS2S_Src [hsm:Instance,s:State,n:String] {
  n in s.(name.hsm) && none = s.(owner.hsm) }
```

- They are then used in a forall-there-exists test;

```
pred S2S_Src [hsm,nhsm:Instance] {
  all s : state.hsm,n : String |
    PS2S_Src[hsm,s,n] =>
      (some s' : State | PS2S_Trg[nhsm,s',n]) }
```

- These tests are *directional* (a dual `S2S_Trg` is defined).

Transformations: Check semantics

- What about when the relations are called with concrete values?
- The standard does not define these semantics;
- We resort to a predicate that also takes elements as input;

```
pred VS2S_Src [hsm,nhsm:Instance,s:HSM/State,s':NHSM/State] {
  all n : String |
    PS2S_Src[hsm,s,n] => PS2S_Trg[nhsm,s',n]) }
```

- They are also directional.

Transformations: Enforce semantics

- We follow the principle of least change, applying the smallest possible update;
- To do so, we need to calculate the distance Δ between Alloy models;
- Since Alloy models are mainly uninterpreted, we resort to the *graph edit distance*;
- Counts the addition, deletion, or relabeling of a vertex or edge.

```
fun Dist_Src [hsm,hsm' : Instance] : Int {
  (#((state.hsm-state.hsm')+(state.hsm'-state.hsm))).plus[
  (#((name.hsm-name.hsm')+(name.hsm'-name.hsm))).plus[
  (#((owner.hsm-owner.hsm')+(owner.hsm'-owner.hsm)))].plus[
  ... }
```

Instances

- Object instances are represented in Alloy as singleton sets belonging to the signature representing its type;

```
one sig S1,S2 extends State {}
one sig T1 extends Transition {}
```

- Their attributes can be simply defined as relations between those signatures;

```
fact {
  name.Src = S1 -> "Simple" }
```

- In order to increase *efficiency*, they can also be defined as upper and lower bounds;

```
fact {
  S1 -> "Simple" -> Src in name
  name in S1 -> "Simple" -> Src + State -> String -> Src' }
```

Execution: Checkonly mode

- Runs the checks in all directions;

```
pred S2S [hsm : Source,nhsm : Target] {
  S2S_Src[hsm,nhsm] && S2S_Trg[hsm,nhsm] }
```

- The scope is the number of existing elements;

```
check { S2S[Src,Trg] && T2T[Src,Trg] }
for 0 but 2 HSM/State,1 HSM/Transition,
          2 NHSM/State,1 NHSM/Transition
```

Execution: Enforce mode

- Asks for consistent models by increasing distance Δ;
- The scope is the number of existing elements plus Δ on the elements of the target model;

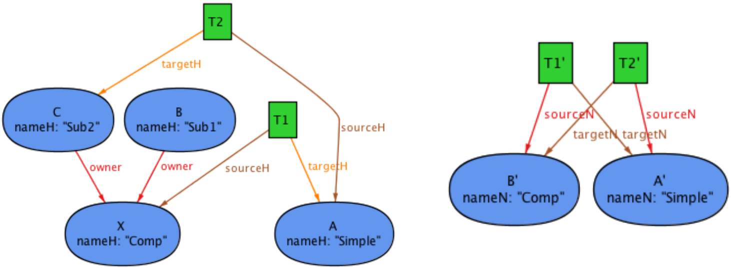
```
run {
  S2S[Src,Trg'] && T2T[Src,Trg'] &&
  Dist_Trg[Trg,Trg'] = 1 }
for 0 but 2 HSM/State,1 HSM/Transition,
          3 HSM/State,2 NHSM/Transition, 2 Int
```

- Guarantees the properties by construction;
- Non-deterministic (for $\Delta \neq 0$).

Recursion

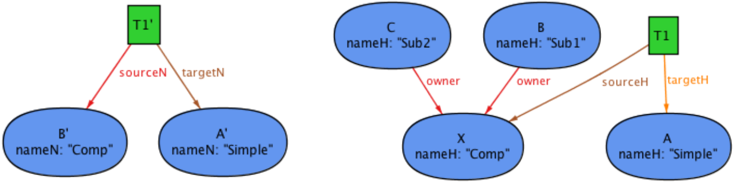
- Alloy does not allow recursive calls;
- Instead, we resort to the transitive closure...
- ... which has just been added to the OCL standard;
- We were able to rewrite the classic recursive QVT examples to use the transitive closure.

Example: Check mode



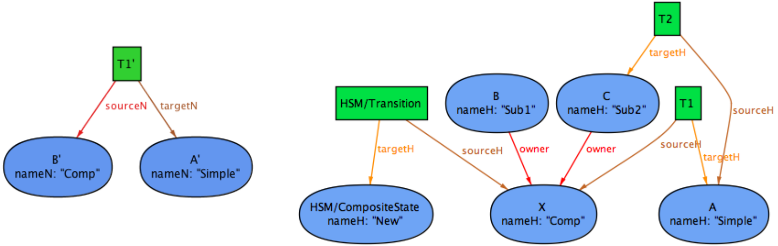

```
check {Check[S,T]}
for 0 but 3 HSM/State,2 HSM/Transition,
          2 NHSM/State,2 NHSM/Transition
```

Example: Enforce mode



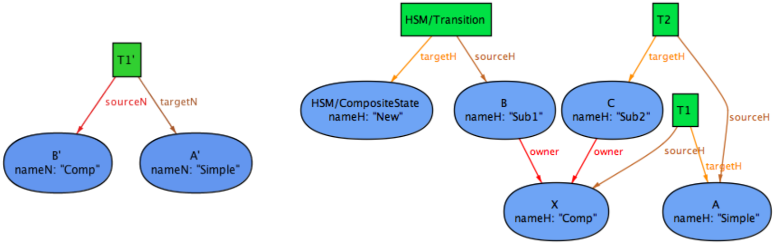

```
run {Check[S',T] & Delta_Src[S,S'] = 3}
for 0 but 3 Int,6 HSM/State,5 HSM/Transition,
          2 NHSM/State,1 NHSM/Transition
```

Example: Non-deterministic




```
run {Check[S',T] & Delta_Src[S,S'] = 5}
for 0 but 4 Int,8 HSM/State,7 HSM/Transition,
          3 NHSM/State,3 NHSM/Transition
```

Example: Non-deterministic




```
run {Check[S',T] & Delta_Src[S,S'] = 5}
for 0 but 4 Int,8 HSM/State,7 HSM/Transition,
          3 NHSM/State,3 NHSM/Transition
```

Conclusions

- We propose and implement a BX framework for QVT where both the models and the transformation can be annotated with generic OCL.

- Open issue: generic mechanism to deal with recursion, either by resorting to the transitive closure or not.
- Distance: how to calculate distances on Integers and Strings?

- Generalization: allow multi-directional transformations;
- Optimizations: infer restrictions from the specifications;
- Non-determinism: is it an issue when we guarantee minimal updates?
- Why not define the consistency relation directly in Alloy?