The Inter-relation Logic

Hui Song

1 Running example

As a compiled engine, the order to execute the top relations is determined at code generation time rather then run-time. For the simple cases where the relations are independent to each other, the top relations can be executed in any order. However, in the practical QVT transformations, the relations often have dependencies between each other, by means of mentioning other relations in one relation's `when` or `where` clauses. When such dependencies exist, executing the top relations in the wrong order may lead to the incorrect transformation result.

We illustrate the dependencies and their influence on the execution order of top relations using the made-up example as follows.

```
transformation sample(source: Meta1, target : Meta2){
 relation A{
  enforce domain source x : X{...} ...
 }
 top relation B {
  enforce domain source x : X{}
  enforce domain source y : Y{} ...
  when { A(x, ...);}
  where{ C(y, ...);}
 }
 relation C {
  enforce domain source y : Y{...} ...
 }
 top relation D{
  enforce domain source y : Y{...}...
  when{ C(y,...);}
 }
}
```

This sample transformation has two top relations, `B` and `C`, and two non-top relations `A` and `D`. Each relation has one or two source domains, and some of them share the common type so that one relation could invoke the other. For the sake of simplicity, we ignore the target domains. Relation `B` has a `when` clause `A(x,...)`, which means before checking if a pair `(x, y)` satisfies `B`, we need to check if `x` satisfies `A`. Or in other word, the satisfaction of `B` depends on the satisfaction of `A`. Similarly, relation `D` depends on `C` because `D` has a when clause `C(y,...)`. In the meantime, `B` has a `where` relation `C(y,...)`, which means that if a pair `(x,y)` satisfies `B`, then we can go on to check and determine the satisfaction of `y` on `C`. From the other perspective, this dependency also means that whether a model element `y` satisfies `C` is not only determined by the constraints defined in C, but also depends on if there exists an element x in type of X and (x,y) satisfies B.

These dependencies determines that when executing the transformation, we must launch the top relation B before D. If we launch D first, then we need to check if y satisfies C. But since B is not executed yet, the evaluation on C(y,...) is not correct, because we do not know if there is an x to make (x,y) satisfy B.

Based on the discussion above, we need a topological sorting of the relations based on their dependencies, in order to determine the order to execute the top relations. However, some QVT relations may not be topologically sortable, because the dependencies among relations may form a circle. Suppose we have another relation E in the sample transformation as follows:

```
transformation sample(source: Meta1, target : Meta2){
 -- Relations A, B, C and D as defined above
 top relation E {
   enforce domain source x : X {}
   enforce domain source y : Y{} ...
   when{ C(y, ...);}
   where{ A(x, ...);}
 }
}
```

Similar to the discussion above, relationE should be executed before B because B{when{A}} and E{where{A}}. But in the same time, B should be executed before E because E{when{A}} and B{where{A}}. Therefore, the two top relations B and C forms a circle, and we cannot decide the execution order between them in the generation time. For the top relations forming a circle, in the generation time, we simply sort them in the same order as they appear in the transformation, and at runtime, we employ a dynamic scheduling mechanism of these relations to ensure the correct behavior. The basic idea is that we execute the top relations according to their textual order, and a relation on a specific tuple of model elements is not ready to go ahead, we hand this relation-element composition at a waiting list, and go back to it when it is ready. Since this dynamic mechanism is time consuming (it actually violates the principle of a compiled QVT engine to decide the execution at compiling time), we also raise a warning on QVT transformation to indicate the circles, so that developers could try to avoid them. It is worth noting that a circle only makes the relations inside it un-sortable, but does not influence the sorting of other relations, nor even the sorting between a relation inside the circle and another one outside. For example, we cannot decide the order of B and E because of the circle, but we are still sure that D should be executed after both of them.

In summary, this paper will present two approaches to the execution of jQVT relations. In the compiling time, we sort the top relations based on their dependencies, which satisfies the the following effects.

1. For any two top relations r_1 and r_2, if the execution of r_2 depends on on the execution of r_1, then the r_1 should locate in front of r_2 in the resulted list.

2. If the first effect is not reachable because there exists two relations r_1 and r_2 that depends on each other, then the order of these two relations is the same as they are defined in the transformation rule. And moreover, both relations should locate after any relation r on which r_1 or r_2 depends but r does not depends on either of them. Similarly, both of them should locate before any relation r' that depends on r_1 or r_2, but neither of them depends on r'.

At runtime, we present an approach to deal with the circled relations by postponing and recording the evaluation of relations that depends on unfinished ones.

2 The problem

This section presents a simple formalization to the problem of relation dependencies in QVT. We name the set of all legal QVT relations as a set $\mathbb{R}$, and any transformation is a sequence of relations, i.e., a set $T \subset \mathbb{R}$, with a document order $\leq \subseteq T \times T$, where $\forall t_1, t_2 \in T, t_1 \leq t_2$ means that t_1 is defined in front of t_2 textually in the QVT rule, or they are the same relation. All the top relations in the transformation forms a subset $T_t \subseteq T$.

The sorting problem on a transformation T is to define a new total order on the top relations $\prec \subseteq T_t \times T_t$, so that in the runtime, the generation engine will execute the top relations according to this new order $\prec$. The sorting is based on the dependency between the relations.

The relations in a transformation may mention each other in the `when` and `where` clauses, forming two relations (the “relations” between sets, not the QVT relations) $\lhd_\uparrow \subseteq T \times T$ and $\lhd_\downarrow \subseteq T \times T$. For any $r_1, t_2 \in T$, $r_1 \lhd_\uparrow t_2$ means that r_1 is mentioned in a `when` clause of t_2, and $r_1 \lhd_\downarrow t_2$ means r_1 mentions t_2 in one of its `where` clause. The triangle points to the relation that should be executed before the other one, i.e., the callee of `when` or the caller of `where`. We use the upward and downward arrows to indicate `when` and `where` intuitively, because according to the coding convention of QVT rules, a `when` is usually placed above the `where` clause in the same relation (such as the sample relation `B`), and it usually points to the relation that are defined before (above) the host one. We also introduce an abstract symbol for direct dependency $\lhd \subseteq T \times T$:

$$\forall r_1, r_2 \in T, r_1 \lhd t_2 \equiv r_1 \lhd_\uparrow t_2 \cup r_1 \lhd_\downarrow t_2$$

From the direct dependency $\lhd$, we get its transitive closure $\lhd^+$, and name it as the *general dependency*:

$$\forall r_1, t_2 \in T : r_1 \lhd^+ t_2 \equiv (r_1 \lhd t_2) \cup (\exists t_3 \in T : r_1 \lhd t_3 \wedge t_3 \lhd^+ t2)$$

The general dependency $\lhd^+$ is a super set of the real mandatory dependency between relations. In other word, $r_1 \lhd^+ t_2$ is a *necessary condition* of the r_1 *must be executed before* t_2. For some pair of generally dependent relations, such as $r_1 \lhd_\uparrow r_2 \wedge r_1, r_2 \in t(T)$, it is still fine if we execute r_2 first, because the engine can evaluate r_1 on site and it does not depends on any other relations. On the other hand, if a relation r_1 must be executed before r_2, such as the sample `B` and `D`, then their must be a chain of `when` and `where` invocations between them, and thus we must have $r_1 \lhd^+ r_2$, because $\lhd^+$ is the transitive closure of the `where` and `when` invocations.

Since the general dependency is a necessary condition for two relations to be executed in the correct order, a topological sort of the top relations (if there exists one) can be correct order $\prec$ to execute the relations. However, for some transformations, the top relations may form a circle, and thus is not topologically sortable.

A circle C formed by the general dependency is a set of relations satisfying $C \subseteq T$ and $\forall r_1, r_2 \in C, r_1 \lhd^+ r_2 \wedge r_2 \lhd^+ r_1$. If two dependent relations are not in a circle, we name their dependency as unidirectional one, noted as $\lhd_u^+$: $\forall r_1, r_2 \in T, r_1 \lhd_u^+ r_2 \equiv r_1 \lhd^+ r_2 \wedge r_2 \not\lhd^+ r_1$. For any circle, all of its member relations have the same unidirectional dependency with any relation outside the circle. Formally speaking, for any circle $C \subset T$ and any relation outside it $r \in T, r \notin C$, if $\exists r' \in C, r \lhd^+ r_1$, then $\forall r'' \in C, r \lhd_u^+ r''$. There cannot be any relation outside a circle which depends on some member in the circle, and in the same time is dependent by some other members in the same circle, because otherwise this relation itself will be one of the circle members. Therefore, although the members inside a circle is not topologically sortable, we can still put them together after all the relations they depend on, and in front of all the relations that depends on them.

From these definition, we define required sorting of top relation as follows. The sorting is a total relation $\prec \subseteq T_t \times T_t$:

$$\forall r_1, r_2 \in T_t : \begin{cases} \text{if } r_1 \lhd_u^+ r_2 \text{ then } r_1 \prec r_2 \\ \text{else if } r_1 \lhd^+ r_2 \wedge r_2 \lhd^+ r_1 \wedge r_1 \leq r_2 \text{ then } r_1 \prec r_2 \\ \text{else if } r_1 \not\lhd^+ r_2 \wedge r_2 \not\lhd^+ r_1 \wedge head(r_1) \leq head(r_2) \text{ then } r_1 \prec r_2 \\ \quad \text{here } head(r) = min_{\leq}(\{r\} \cup \{r' | r' \lhd^+ r \wedge r \lhd^+ r'\} \cap T_t) \\ \text{else } r_2 \prec r_1 \end{cases}$$

If two top relations have a unidirectionally dependency, they are sorted according to this dependency. If they belong to the same circle, then they are sorted according to their document order. If they do not have any dependency, we also sort them based on their document order. However, to make sure the circled relations are placed together, if a relation belongs to a circle, we use the minimal document location from the circle members rather then its own location. Intuitively, if a relation has no dependency with a circle, then it will be sorted before the circle members if it is originally defined in front of any of the circle members, otherwise it will be sorted behind all the circle members.

It is easy to see that $\prec$ is a total order, and the sorting according to it satisfies the requirement defined in the last section.

3 Sorting the top relations

Algorithm 1 describes the algorithm to sort top relations. The input of the algorithm is the jQVT transformation as a set of relations, and the output is a sequence of the sorted top relations.

Algorithm 1: Top relation sorting

```
In: A transformation $T$, its top relations $T_t$
Out: A sequence $T_{sort} \in T_t*$ of sorted top relations
1  pool $\leftarrow T$ ;
2  repeat
3      foreach $r \in$ pool do
4          if $c \leftarrow$ FindCircle($r, \{\}$) then
5              $fk \leftarrow$ FakeRelation($c$) ;
6              pool $\leftarrow$ pool $- c \cup \{fk\}$;
7              break;
8  until No $r \in$ pool leads to a circle ;
9  $T_{sort} \leftarrow \{\}$ ;
10 repeat
11     let head $\leftarrow \{r \in$ pool$|\neg(\exists r' \in Pool)[r' \lhd r]\wedge\}$ ;
12     pool = pool $-$ head ;
13     subseq $\leftarrow$ DocSort(head) ;
14     foreach $r \in$ subseq do
15         if $r \in T_t$ then $T_{sort} += r$ ;
16         else if $r$ is FakeRelation then
17             $T_{sort} +=$ DocSort($\{r' \in r$.circle$|r' \in T_t\}$) ;
18 until pool $= \{\}$ ;
19 function FindCircle (cur $\in T_t$, trace $\in T_t*$) $\rightarrow$ circle begin
20     if cur $\in$ trace then
21         $i \leftarrow$ Index(trace, cur) ;
22         circle $\leftarrow \{$trace$[k]|i \leq k \leq$ trace.len$\}$ ;
23         return circle;
24     else
25         foreach $r \in$ pool, cur $\lhd r$ do returnFindCircle($r$, trace $+ r$) ;
26 end
```

We first put all the relations into a temporal relation pool (Line 1), and then we keep taking one relation r from the pool to see if there will be any circle in the path starting from r (Lines 3-7). If there is a circle c, we remove all the relations inside c from the pool, construct a `FakeRelation` from this circle, and put it back to the pool (Lines 4-7). A fake relation is a made-up relation in represent of a circle of relations. Act as a relation, it supports the document order and dependency between real relations:

$$\begin{aligned} r \lhd fk &\equiv (\exists r' \in fk.\texttt{circle})[r \lhd r'] \\ r \leq fk &\equiv (\exists r' \in fk.\texttt{circle})[r \leq r'] \end{aligned} \tag{1}$$

How to find the circle from r is defined in the function `FindCircle`(Lines 19-25), which is simply a depth-first search starting from any relation `curr`, go along the dependable relations, and record the traces. If the search reaches a relation already in the trace, there is circle found. We keep this loop until no relation in the pool will lead to a circle. That means the relations in the pool are acyclic. After that, we perform a topological sorting on this pool and break the circles, in order to get a sorted top relation sequence (Lines 10-18). The topological sorting process is as follows. We first retrieve a set of relations (`head`) that do not depends on any other relations in the pool, and remove these relations from the pool (Line 12). After that, we take a relation one after another according to the document order. For each relation r, if it is a top relation, we put it into the relation sequence (15), otherwise, if it is a fake relation, we put the top relations inside the circle also according at the end of the sequence, also following the document order (Lines 16-17).

4 Circle handling at runtime

In the runtime, the generated jQVT engine will evaluate the top relations one after another according to the order we calculated before generation. For each relation, we assemble all the possible compositions from the source model elements as the domains of the relation, and evaluate each of these compositions on the relation. If the relations do not form any circle, this evaluation order guarantees that before any relation is being evaluated, all relations it depends on have been evaluated, and thus their results are determinate. However, if there are circles, this assumption is not guaranteed, and it is possible that when a relation is under evaluation, some of its `when` criteria does not have a determinate result. To handle this, we employ a runtime circle handling mechanism to postpone the evaluation of these relations until all the relations it depends on have been evaluated. The mechanism has two phases as follows.

When evaluating a relation r on domains $e_1, ...e_m$, we first check if all of its `when` clauses are true. If they are, we go ahead on the with the evaluation. Otherwise, if there is one clause mentioning a relation r' with domains $e_1, ..., e_n$ returns false, there are two potential reasons. The first is that r' does not hold on $e'_1, ..., e'_n$, and we don't need to go on with the evaluation of r. The second is that r' *may* hold on $e'_1, ..., e'_n$, but the top relation that may trigger the evaluation of r' on these domains has not been evaluated yet. Since we do not know the exact reason, we postpone the evaluation on r by adding a three-tuple of r, $e_1, ...e_m$ and $e'_1, ..., e'_n$ into the waiting list of r'.

In the meantime, after a relation r' is successfully evaluated, we check its waiting list, and retrieve all the relations waiting for this r', and retrieve the three-tuple that matches the domain list of the current evaluation. For each of such three tuple, we re-evaluate the relation r with its domain list $e_1, ..., e_m$. In this way, the evaluation of r on $e_1, ..., e_m$ is postponed until its dependent relation r' is evaluated on $e'_1, ..., e'_n$, and influence of the circle is eliminated.