

Transformation Mapping (KTM)

KTM - KIELER Transformation Mapping

Deprecated since 0.12

This article is deprecated. The described features are no longer available in current releases.

KTM was redesigned is now available as KITT included in KiCool.

Topics

- [Transformation Tree Model](#)
- [Extensions](#)
- [Implementation Details](#)
- [Example](#)
- [Visualisation](#)

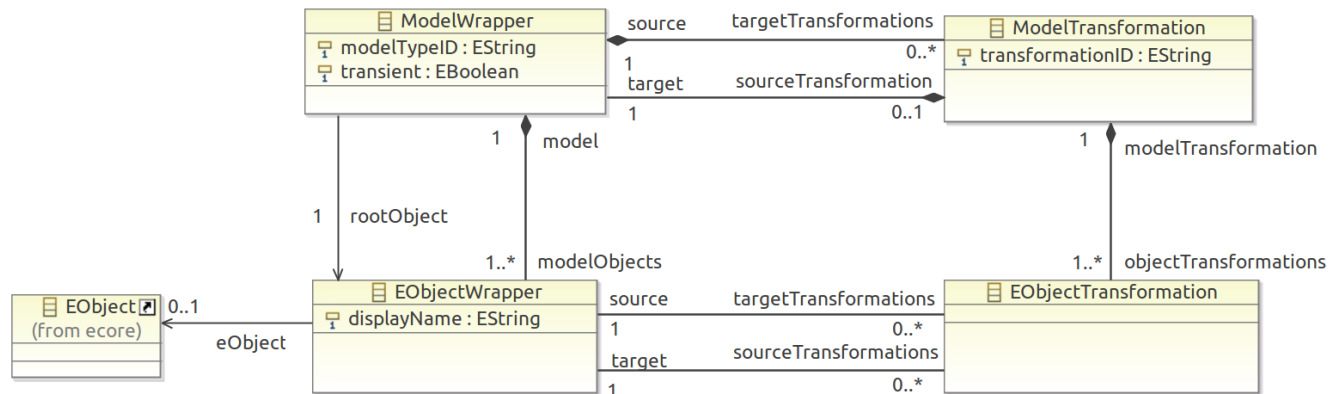
This subproject provides a tracing mechanism for arbitrary model-elements across multiple model transformations, based on EMF.

The main propose of KTM is to allow bidirectional information transfer between abstract models and their resultant transformed models.

Transformation Tree Model

To offer a mapping between model-elements during multiple transformations KTM introduces a model called TransformationTree to represent these relations.

It is based on an EMF-Metamodel.

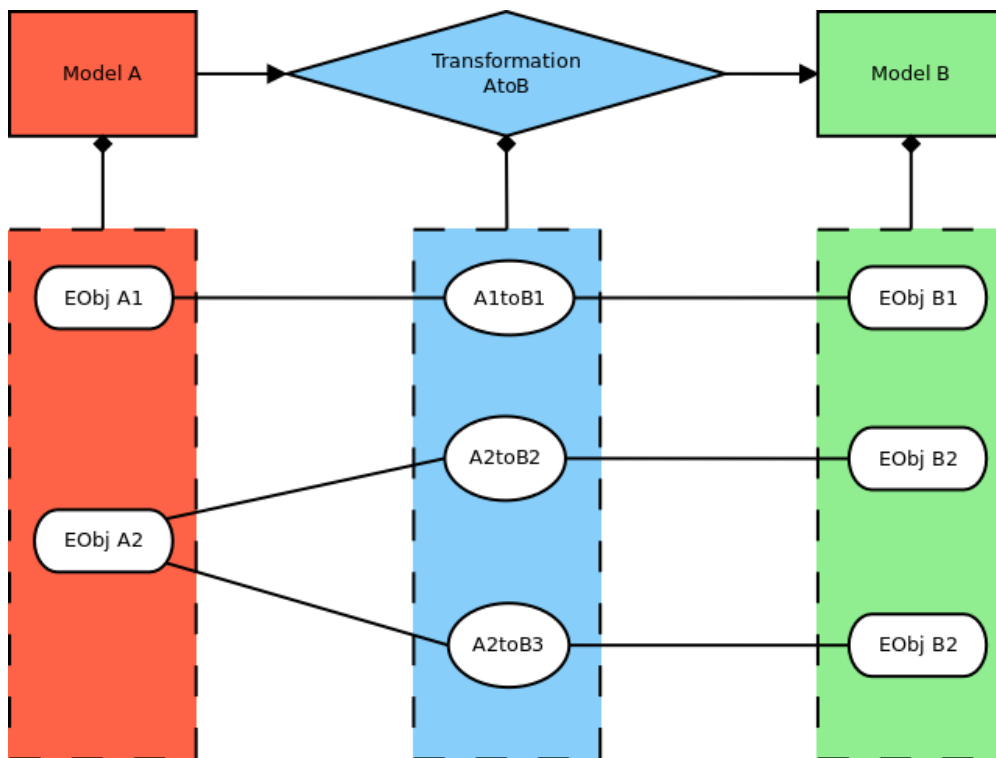


The structure of the model can be separated into two parts.

First part (upper half) is a tree of transformations. Each **ModelWrapper**-class is a representation of a model which is transformed. So **ModelWrapper** are nodes and **ModelTransformations** are edges. Thus the **ModelWrapper** representing the initial-source-model of all transformation is also the root of a **TransformationTree**-model.

Second part (lower half) is object-mapping. Instances of models contain **EObjects** as their elements, which are represented by **EObjectWrapper**-class in this metamodel. The **EObjectWrapper** of two models are connected with **EObjectTransformations**-class to express their origination relationship in corresponding model transformation.

An abstract example of an instance of this model:



Extensions

Two classes are provided by this project to extend functionality of the core model.

TransformationMapping ([JavaDoc](#))

The main propose of this class is generation of a object-mapping during transformation process.

Therefor it provides different functions for incremental registering of single parent-child-relations between EObjects.

Furthermore, the extension allows to extract the mapping and check completeness of mapped elements against content of transformed models.

TransformationTreeExtensions ([JavaDoc](#))

This class provides all functionalities to easily traverse and search in a TransformationTree.

Furthermore, it allows to modify trees by creating, deleting or appending new transformations and transformed models.

Additionally this extension provides functionality to extract a concrete mapping between two arbitrary model instances from a TransformationTree.

Implementation Details

- All references to EObjects in EObjectWrapper are references to a copy of the original EObject. This allows to represent immutable mapping. To reidentify corresponding EObjects TransformationTreeExtensions provides search functions which will check for structural matching models.
- Models in TransformationTrees may be transient. This indicates that all references to EObjects in all Elements of the transient model are removed. Thus these models can't be source of a new appended transformation and can not be associated with it's original model. The main propose of this feature is to improve scalability of TransformationTrees by removing unnecessary references to internal model, but preserve traversing functionality of the object-mapping.
- Mappings can be incomplete causing resulting transformation tree to be incomplete. A incomplete tree does not represent every object in a model with a corresponding Element. This may break some paths of element transformations, but allows to omit model-immanent objects like annotations from mapping. TranformationMapping extension provies a function to check completeness of mapping against its models.

Example

In this example we will perform some transformations on SCCharts.

The source chart is a ABO, the "Hello World" of SCCharts.

ABO is already a CoreSCChart, so we will perform normalization and a transformation to SCG.

Creating Mapping during Transformation

In order to note every single element transformation of a model transformation, we use the TransformationMapping extension.

After each creation of new Objects for transformed model the mapping must be updated with it's origin information.

The codeblock blow show a snipped of SCChartCoreTransformation with additional mapping registration.

transformTriggerEffect CodeSnipped

```
...
@Inject
extension TransformationMapping

...

// NEW - Mapping access delegation
def extractMapping() {
    extractMappingData;
}

...

//-----
//--          S P L I T   T R A N S I T I O N          --
//-----
// For every transition T that has both, a trigger and an effect do the following:
//   For every effect:
//     Create a conditional C and add it to the parent of T's source state S_src.
//     create a new true triggered immediate effect transition T_eff and move all effects of T to T_eff.
//     Set the T_eff to have T's target state. Set T to have the target C.
//     Add T_eff to C's outgoing transitions.
def Region transformTriggerEffect(Region rootRegion) {
    clearMapping; //NEW - clear previous mapping information to assure a single consistent mapping
    // Clone the complete SCCharts region
    var targetRootRegion = rootRegion.mappedCopy; //NEW - mapping information (changed copy to mappedCopy)
    // Traverse all transitions
    for (targetTransition : targetRootRegion.getAllContainedTransitions) {
        targetTransition.transformTriggerEffect(targetRootRegion);
    }
    val completeness = checkMappingCompleteness(rootRegion, targetRootRegion); //NEW - DEBUG
    targetRootRegion;
}

def void transformTriggerEffect(Transition transition, Region targetRootRegion) {
    // Only apply this to transition that have both, a trigger (or is a termination) and one or more effects
    if (((transition.trigger != null || !transition.immediate || transition.typeTermination) && !transition.
effects.nullOrEmpty) ||
        transition.effects.size > 1) {
        val targetState = transition.targetState
        val parentRegion = targetState.parentRegion
        val transitionOriginalTarget = transition.targetState
        var Transition lastTransition = transition
        val firstEffect = transition.effects.head
        for (effect : transition.effects.immutableCopy) {
            // Optimization: Prevent transitions without a trigger
            if(transition.immediate && transition.trigger == null && firstEffect == effect) {
                // skip
            } else {
                val effectState = parentRegion.createState(GENERATED_PREFIX + "S")
                effectState.mapParents(transition.mappedParents); //NEW - mapping information
                effectState.uniqueName
                val effectTransition = createImmediateTransition.addEffect(effect)
                effectTransition.mapParents(transition.mappedParents); //NEW - mapping information

                effectTransition.setSourceState(effectState)
                lastTransition.setTargetState(effectState)
                lastTransition = effectTransition
            }
        }
        lastTransition.setTargetState(transitionOriginalTarget)
    }
}
```

Create TransformationTree

The following code will now perform each transformation stepwise and updates a transformation tree each step.

Transform and create TransformationTree

```
aboSplitTE = SCCtransformation.transformTriggerEffect(abo);

ModelWrapper aboSplitTEModel =
    transformationTree.initializeTransformationTree(SCCtransformation.extractMapping(), "TriggerEffect",
    abo, "coreSCChart", aboSplitTE, "coreSCChart-splitTriggerEffect");

aboNormalized = SCCtransformation.transformSurfaceDepth(aboSplitTE);

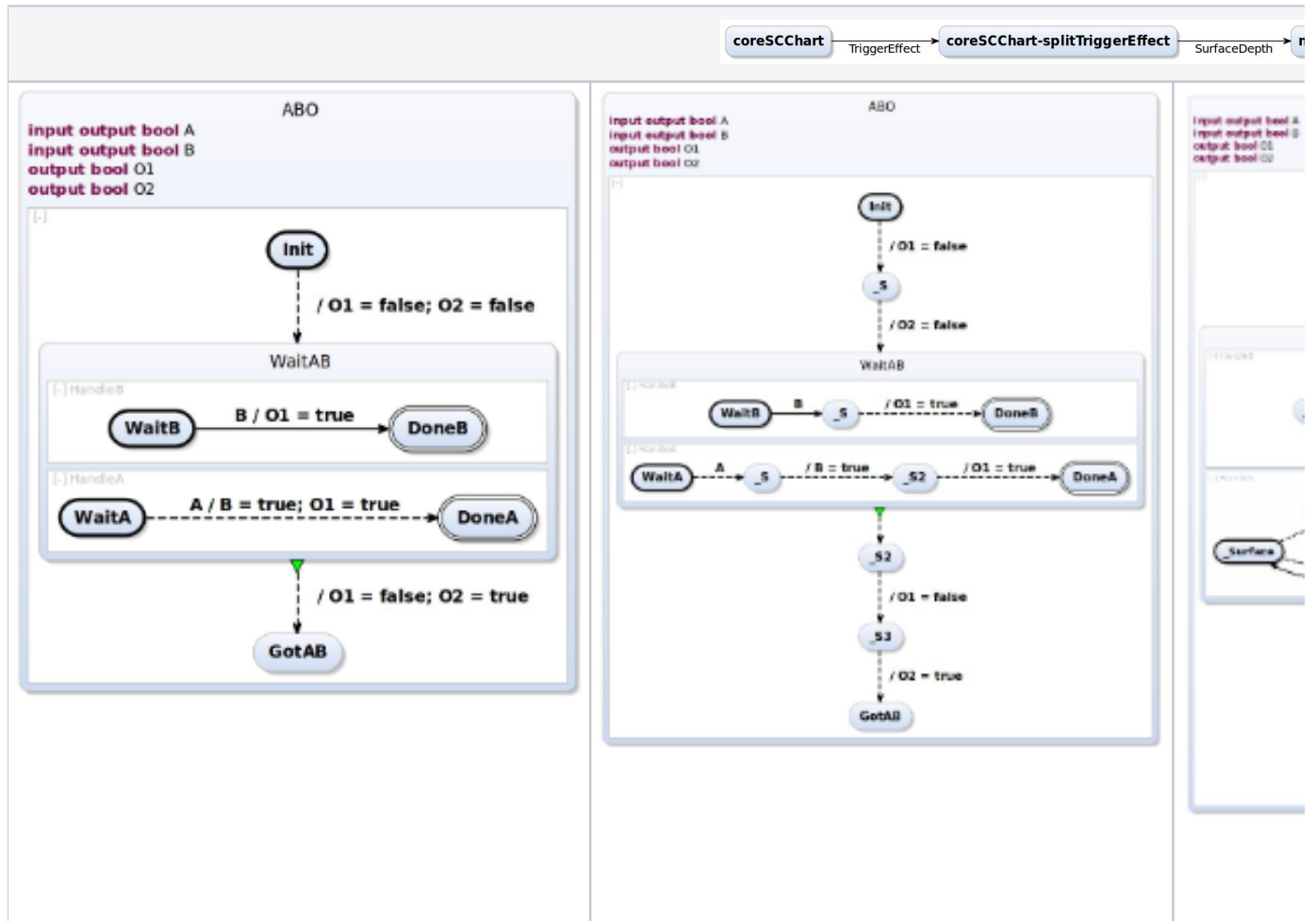
ModelWrapper aboNormalizedModel =
    transformationTree.addTransformationToTree(SCCtransformation.extractMapping(), aboSplitTEModel,
    "SurfaceDepth", aboSplitTE, aboNormalized, "normalizedCoreSCChart");

aboSCG = SCGtransformation.transformSCG(aboNormalized);

ModelWrapper aboSCGModel =
    transformationTree.addTransformationToTree(SCGtransformation.extractMapping(), aboNormalizedModel,
    "SCC2SCG", aboNormalized, aboSCG, "SCG");

tree = transformationTree.root(aboSCGModel);
```

The resulting TransformationTree has following structure and representing each step and model of the transformation.



Furthermore the TransformationTree now contains mapping information for the whole transformation chain.

Now we can use an additional feature of KTM, the resolving of mappings between arbitrary models.

The following code has starts with an instance of the initial ABO SCChart and SCG, along with the TranformationTree above.

resolveMapping

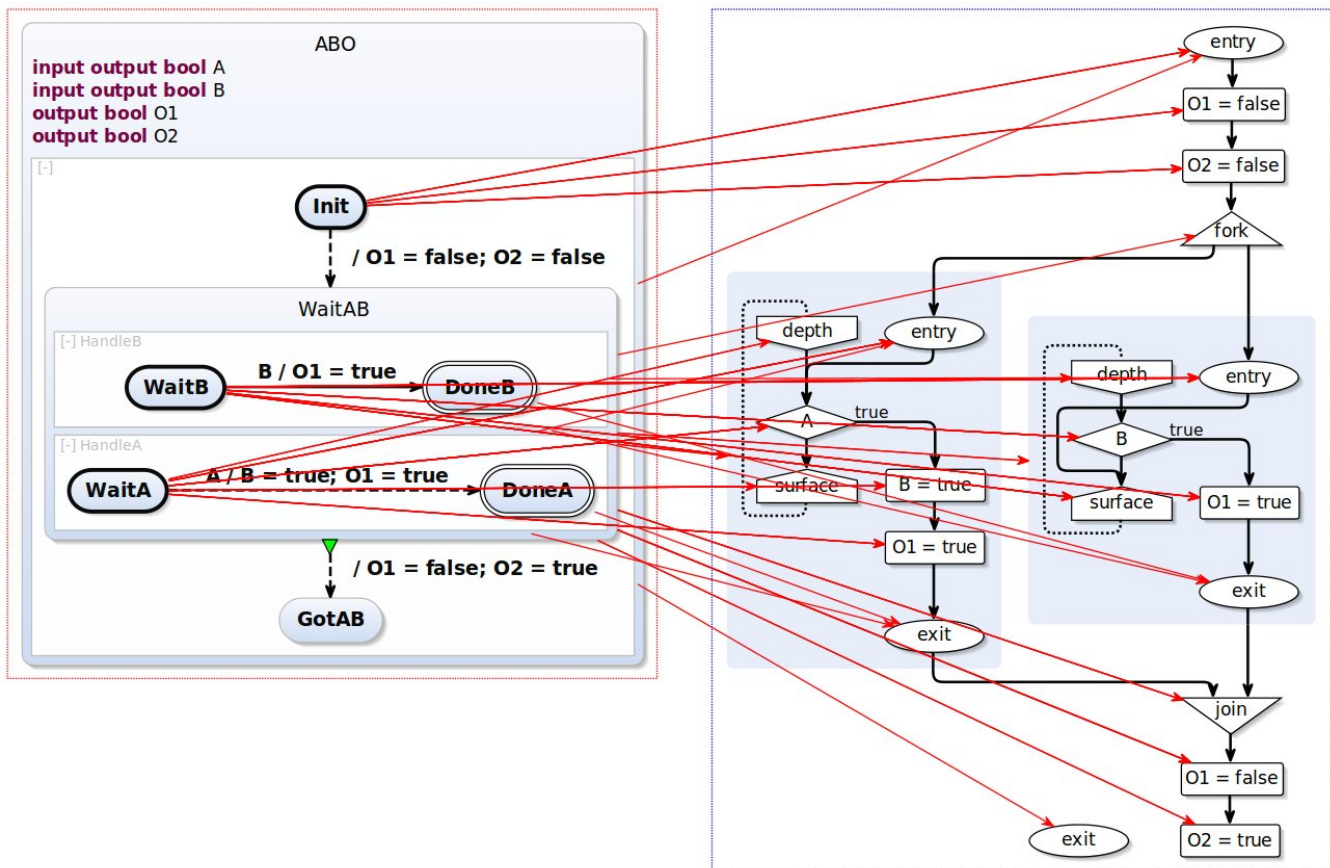
```
@Inject
extension TransformationTreeExtensions

//Find nodes of model instances in tree
val aboSCCModelWrapper = transformationTree.findModel(abosCC,"coreSCChart");
val aboSCGModelWrapper = transformationTree.findModel(abosCG,"SCG");

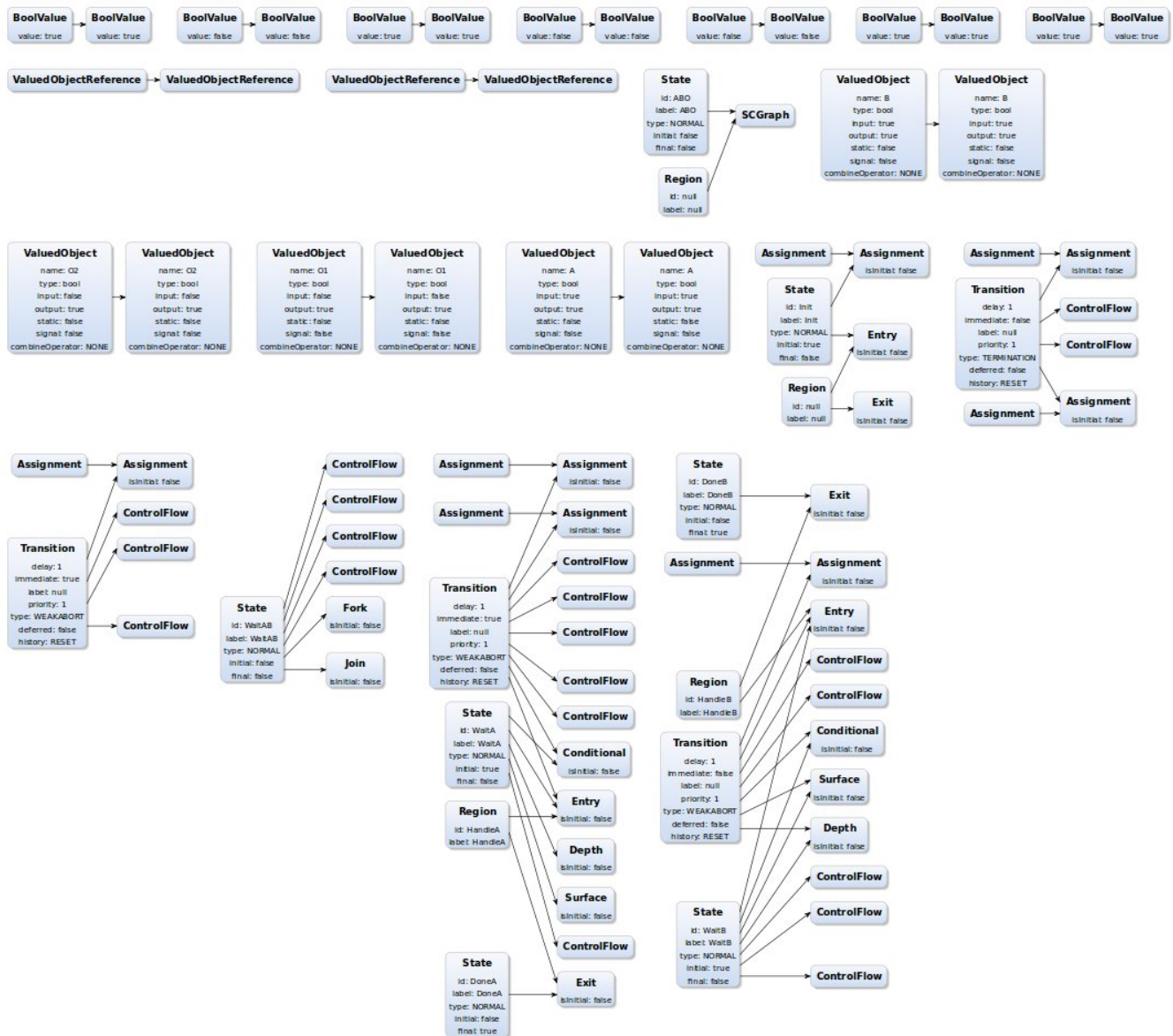
//resolve
val mapping = resolvemapping(abosCCModelWrapper, abosCC, abosCGModelWrapper, abosCG);
```

The returned mapping is a multi mapping between all object in abosCC and their resulting objects in abosCG.

This mapping can now displayed in models or used for various information propagation between elements of the models.



Also a more detailed view is available, showing all EObjects relation.



Visualisation

If you have a TransformationTree file (.kmt) you can open a KLighD visualisation by right-clicking on file in project-tree and selecting 'Open Transformation Tree'.

Diagram Options

Model Visualisation: If enabled tries to visualize selected models with KLighD else a EObject-representation is created.

EObject Attributes: If enabled shows Attributes of EObject in EObject-representation.

Selective mapping edges: If enabled shows only selected mapping edges.

Interaction

CTRL+CLICK: Selects a Node in TransformationTree as source and displays its represented model.

SHIFT+CLICK: Selects a Node in TransformationTree as target, displays both models and the resolved mapping as edges (currently only between States /Regions).

If Selective selective mapping edge is enabled no mapping edges are displayed. If you select (**CLICK**) an element in one of the two model its relation to corresponding element is displayed. You can multi-select with **CTRL+CLICK** or deselect by clicking on an edge.

