SiDiff 2.0 – Introduction to modelmanagement

Oliver Scharmann

25. November 2009

Inhaltsverzeichnis

Dokumenthistorie

Dieses Dokument wird fortlaufend gepflegt. Die nachfolgende Tabelle gibt eine Übersicht über die Änderungen in einzelnen Versionen.

Datum	Änderungen
9.9.09	erste Version
22.9.09	Ergänzungen und Systemüberblick
27.10.09	Strukturierung und Abbildungen

Geplante Änderungen

Folgende Änderungen sind bereits absehbar, wurden im aktuellen Dokument aber noch nicht umgesetzt:

1 Vorwort

Der Inhalt folgender Quellen über das Eclipse Modeling Framework im allgemeinen und Metamodelierung im speziellen wird im folgenden Vorrausgesetzt:

- Kapitel 2 aus: Dave Steinberg et al., EMF: Eclipse Modeling Framework, Addison-Wesley Longman, ISBN 978-0-32133-188-5, 2009
- Abschnitt EMF aus dem Skript zur Vorlesung Softwaretechnik I
  http://pi.informatik.uni-siegen.de/Mitarbeiter/kehrer/lehre/ws08/st1/emf
- Skript über Metamodelle
  http://pi.informatik.uni-siegen.de/kelter/lehre/08w/lm/lm_mm_20081124_a5.pdf
- EMF Homepage
  http://www.eclipse.org/modeling/emf/

2 Motivation

Im Kontext von SiDiff stellt sich die Herausforderung nahezu beliebige Datenstrukturen verarbeiten zu müssen. So besitzt jeder zu unterstützende Dokumenttyp ein eigenes Metamodell, sowie dessen Umsetzung in Form einer entsprechenden, streng strukturierten Datenstruktur. Zusätzlich können die Werkzeugeigenen Metamodelle nur selten zu direkten Verarbeitung geeignet oder bekannt. Somit ist es notwendig für jeden Dokumentyp ein entsprechendes Metamodell zumindest implizit zu entwickeln.

Eine einfache Methode, um beliebige Datenstrukturen zu verarbeiten, ist es diese als allgemeine Graphstruktur zu interpretieren und zu verarbeiten. Dieses Vorgehen besitzt jedoch den Nachteil, dass

- das Wissen über die Dokumentstruktur (Metamodell) lediglich implizit vorliegt
- eine laufzeitintensive, dynamische Graph Implementierung verwendet werden muss
- eine dynamische Graph-Instanz aus den Daten erzeugt werden muss
- Fehlerhafte Dokumente unerkannt bleiben
- uvm.

Das Eclipse Modeling Framework (EMF) stellt eine Metasprache und Werkzeuge bereit, die es ermöglicht Metamodelle zu Formulieren und anschließend entsprechende Instanzen zu erzeugen und zu verarbeiten. Dabei kann nativer Code für die Datenstrukturen generiert und verwendet werden, was dynamische Datenstrukturen überflüssig macht.

- Dokumentation des Datenstruktur durch explizites Metamodel

- Graphartige, reflektive Schnittstelle.
- Statische (durch automatisiert generierten Modellcode) und dynamische Realisierung (durch dynamische Interpretation von Metamodellen) von Modelinstanzen möglich
- Persistenz durch De-/Serialisierung in XML
- EMF/Ecore findet bereits breite Anwendung im Werkzeugbau; Entsprechende Werkzeuge können somit nativ unterstützt werden.

3 Verarbeitung von Modellen in SiDiff

Mit SiDiff werden die Daten von Anwendungen (z.B. Modellierungswerkzeuge, UML-Diagramme oder Simulationstools) in unterschiedlichen Versionen verglichen. Um diese Daten verarbeiten zu können, wird ein Metamodell erstellt, welches die Daten für den Vergleich und die Berechnung der Differenzen strukturiert ablegt. Dieses Metamodell kann man auf unterschiedliche Weise spezifizieren. In EMF erstellt man ein Ecore-Modell, welches das Aussehen eines UML-Klassendiagramms besitzt aber über eine eigene Typwelt verfügt, und die Serialisierung und Deserialisierung der Anwendungsdaten ermöglicht. Als Format verwendet EMF *XML Metadata Interchange (XMI).*
Für die Verwendung einer Anwendung mit SiDiff, sollte diese ihre Daten optimalerweise im XML-Format exportieren können, weil so nur ein XSLT-Skript geschrieben werden muss, um die Daten in das Format des Metamodells zu überführen. Verwendet das Werkzeug jedoch ein proprietäres Format, so muss ein Parser erstellt werden, welcher die Syntax der Daten analysiert und über Methodenaufrufe des aus dem Metamodell generierten Javacodes die Instanzen der Datenstrukturen anlegt.
EMF kann aus den per XSLT transformierten und in XMI gespeicherten Daten über die Deserialisierung automatisch die Laufzeitdatenstrukturen im Speicher aufbauen. Anschließend können die Laufzeitdatenstrukturen an den SiDiff-Algorithmus weitergereicht werden, oder alternativ im Modelstorage abgelegt werden. Zusätzlich bietet EMF eine generische Schnittstelle (reflective API) für den Zugriff auf die Elemente des Metamodells. Durch die Verwendung dieser Schnittstelle kann der SiDiff-Algorithmus unabhängig vom Metamodell arbeiten.

4 Metamodellierung

Eine *.ecore-Datei entspricht einem Metamodell. In SiDiff beschränken wir uns auf Strukturierungselemente von Ecore d.h. einige wenige EMF-Elemente, um Datenmodelle zu abzubilden.Hierbei dient die Vererbung als Hilfsmittel um Attribute bzw. Referenzen von anderen Klassen zu übernehmen. („Abkürzung“)
Pakete werden nur zur Strukturierung und ohne weitere semantische Bedeutung verwendet. Wir empfehlen die *explizite* Modellierung eines Wurzelelements.

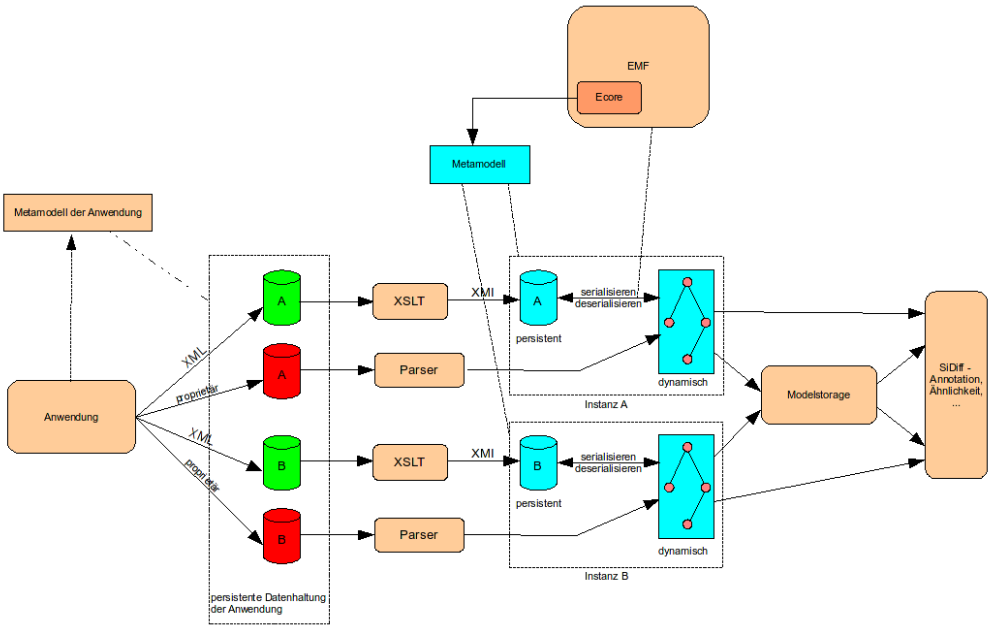

Abbildung 1: Systemüberblick

4.0.1 Modellelemente

EClass Modellelemente (auch solche, die in Diagrammen als Kanten dargestellt werden)

EAttribute Attribute (Eigenschaften von Knoten)

EReference Beziehungen von Modellelementen bzw. Referenzen auf andere Modellelemente. `EReference` ist gerichtet. Um bidirektionale Beziehungen abzubilden werden zwei gegenläufige `EReference`s benötigt, die sich jeweils als `EOpposite` eingetragen haben.

EAnnotation Metainformationen für Modellelemente. `EAnnotations` können genutzt werden, um Teile des Algorithmus bzw. der Komponenten zu steuern.

4.0.2 Kantentypen (aus Entwicklersicht)

Wir unterscheiden ferner zwischen folgenden Arten von Kanten. Die Art wird in der `EdgeSemantic` ausgedrückt.

Submodel Referenzen auf ein Submodell (gekennzeichnet durch eine EAnnotation "SUBMODEL" an der Kante)

Nesting Baumkanten vom Vater zum Kind

Parent Baumkanten vom Kind zum Vater

Reference alle Kanten, die keine Baumkanten sind

Incoming Referenzen, die im Metamodel mit der EAnnotation "INCOMING" markiert sind. Diese Kanten sind eigentlich nicht Bestandteil des Metamodells; sie wurden nur als Rückwärtskanten eingefügt, um einer *eingehenden Kante* entgegenlaufen zu können.

Outgoing Referenzen, die nicht mit INCOMING markiert sind (Standardfall)

4.0.3 Annotationen

Annotation	Nutzendes Bundel	Elemente
Submodel	„Edge Semantik“	EReference
Incoming	„Edge Semantik“	EReference
core.difference.differencemodel	Es wird keine Differenz für das annotierte	Element ausgegeben
NoHash	core.annotators.hashing	Attributwert bzw. Referenzzie in die Hashwertberechnung e
Path	core.annotators	Attributwert wird zum bilden verwendet.
AbsolutePosition	core.difference.differencemodel	Differenz wird auf Basis der Ab gebildet. (Pflicht bei „UpperB
RelativePosition	core.difference.differencemodel	Differenz wird auf Basis der Re gebildet. (Pflicht bei „UpperB

4.0.4 Best Practice

4.1 Werkzeugt und Generierungsprozess

Zur komfortablen, graphischen (Weiter-)Entwicklung der Metamodelle wurde daher der IBM Software Modeler in den Buildprozess einbezogen.

4.1.1 Import von Software Modeller Modellen

Ant und Imprt Skript.

5 Das Arbeiten mit Modellen

Model = Metamodell + Instanz; Mussen verwaltet werden.

5.1 ModelStorage (org.sidiff.common.emf)

Der Modelstorage dient als Zentraler Funktionseinheit für das Laden und den Import von Modellen. Gleichzeitig ermöglicht der Modelstorage aber auch den einheitlichen Zufriff auf beliebige, zu verarbeitende Modell.

5.2 Arbeiten mit Metamodellen

- Wie komme ich an das Metamodell? (EPackage.Registry..., obj.eClass())

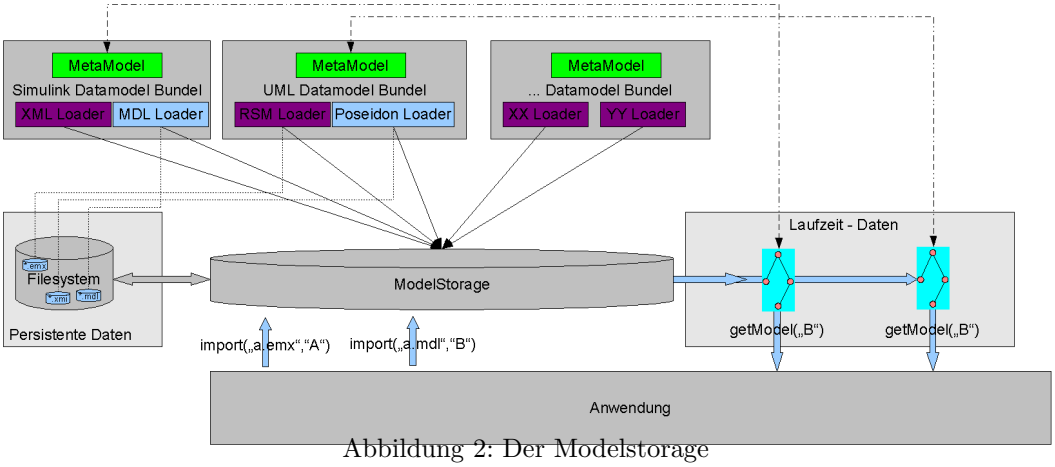

Abbildung 2: Der Modelstorage

5.3 Arbeiten mit Instanzen

- Ein Modell wird durch eine Resource repräsentiert
- EMF-eigener Deserialisierungs-Mechanismus setzt bestz. Format voraus
- Externe Darstellung muss entsprechend transformiert werden (z.b. mit XSLT) (mehrere Varianten, Dokument von ETAS wiederverwenden)
- Nutzung der reflektiven Schnittstelle (eGet()), Verweis auf genaue Stelle im Buch

5.4 Navigation mit Pfaden

Austauschbare Path Engine ...

5.5 EMFUtil

Mit `EMFUtil` werden allgemeine, häufige EMF Operationsfolgen zu einzelnen zusammengefasst.

Objektliste einer Referenz abfragen Mit `EMFUtil.getObjectListFromReference(EObject object, EReference reference)` kann für ein Objekt `object` eine Referenz `reference` abgefragt werden. Das/die Referenzziel(e) wird als Liste vom Typ `EObjec` zurückgegeben.

Mit `EMFUtil.fillObjectListFromReference(List<EObject> result, EObject object, EReference reference)` kann dieselbe Abfrage ausgeführt werden, jedoch wird hier die übergebene Liste `result` gefüllt anstatt eine neue anzulegen.

AllContents als Liste Da `EObject.eAllContents()` bzw. `Resource.getAllContents()` nur Iteratoren zurückliefern, aber oftmals Objektlisten benötigt werden, kann mit `EMFUtil.getEAllContentsAsList(...)` den Gesamtinhalt eines Objekts oder eine Ressource als Liste abfragen.

Menge der Elemente in AllContents Da `EObject.eAllContents()` bzw. `Resource.getAllContents()` nur Iteratoren zurückliefern, aber oftmals die Anzahl der Objekte benötigt wird, kann mit `EMFUtil.getEAllContentsSize(...)` die Anzahl abgefragt werden.

Objekt-ID abfragen Mit `EMFUtil.getEObjectID(EObject eobj)` kann die ID eines Objekts abgefragt werden. Wir erwarten, dass jede Metaklasse ein Stringattribut als ID definiert.

Attributwert abfragen Mit `EMFUtil.getEObjectsAttribute(EObject eobj, String attributeName)` kann direkt der Wert eines Attributs abgefragt werden.

5.5.1 EMFMetaAccess

Mit `EMFMetaAccess` können Abfragen auf dem Metamodell getätigt werden.

Abfrage von Meta-Klassen `EMFMetaAccess` bietet mehrere Methoden an, um Meta-Klassen abzufragen.

`EMFMetaAccess.getMetaObjectByName(EObject object, String type)` gibt die `EClass` vom Typ `type` zurück. Hierbei wird im Metamodell des gegebenen `object` gesucht. Außerdem wird auch in den Metamodellen potenzieller Supertypen der Metaklasse von `object` gesucht.

`EMFMetaAccess.getMetaObjectByName(String packageNS, String type)` gibt den `EClassifier` vom Typ `type` aus dem Metamodell mit dem gegebenen Paketnamensraum `packageNS` zurück. `packageNS` ist hierbei der namespace URI.

`EMFMetaAccess.getMetaClassesForPackage(String packageNS)` gibt eine Liste alle Metaklassen eines Metamodells zurück.`packageNS` ist hierbei der namespace URI des Metamodells.

Eine Liste von Metaklassen kann man sich anhand einer kommaseparierten Liste von Namen geben lassen. Die Funktion wird durch `EMFMetaAccess.getMetaObjectListByString(String packageNS, String commaSepList, Class<T> type)` angeboten. `type` gibt mit `EClass.class` oder `EClassifier.class` an, auf welchen Typ die Ergebnisliste gecastet wird.

Abfrage von Referenzen Eine bestimmte Referenz einer Metaklasse kann mittels `EMFMetaAccess.getMetaReferenceByName(String packageNS, String type, String referenceName)` abgefragt werden.

Alle Referenzen einer Klasse kann mit `EMFMetaAccess.getReferences(EClass eClass)` abgefragt werden. Referenzen zwischen Eltern und Kindknoten werden hierbei ausgeschlossen.

Um bei der Abfrage von Referenzen nur nach solchen mit einer bestimmten Semantik zu fragen, kann man `EMFMetaAccess.getReferences(EClass eClass, EdgeSemantic semantic)` aufrufen.

Referenzen die auf Kindknoten zeigen können mit `EMFMetaAccess.getChildrenReferences(EClass eClass)` abgefragt werden.

Mit `EMFMetaAccess.getReferencesByNames(EClass eClass, String names)` kann eine Menge von Referenzen einer Metaklasse abgefragt werden. `names` ist entweder eine kommaseparierte Liste von Namen oder ein regulärer Ausdruck.

Mit `EMFMetaAccess.translatePath(EClass eClass, String path)` kann ein Pfadausdruck in eine `EMFPath` übersetzt werden.

Abfrage von Attributen Mit `EMFMetaAccess.getAttributesByRegEx(EClass eClass, String regex, boolean regexExpectedResult)` kann man zu einer Metaklasse eine Menge an Attributen abfragen, auf deren Namen der reguläre Ausdruck `regex` matched. Mit `regexExpectedResult` kann das erwartete Ergebnis der Ausdrucksprüfung festgelegt werden, um Attribute in die Ergebnisliste zu übernehmen: `true` heißt der reguläre Ausdruck muss erfüllt sein, `false` heißt, er darf nicht erfüllt sein.

5.5.2 EMFModelAccess

Mit `EMFModelAccess` werden komfortable Abfragen auf Modellinstanzen angeboten. Abfragen, die mit dem allgemeinen Ecore-Operationsumfang getätigt werden können, sollen hier jedoch nicht unnötig gekapselt werden. `EMFModelAccess` konzentriert sich nur auf Abfragen und Funktionen, die über den allgemeinen Ecore-Operationsumfang hinausgehen.

Nachfolgend stellen wir häufige Abfragen vor, ungeachtet dessen, ob sie mittels `EMFModelAccess` oder mit dem allgemeinen Ecore-Operationsumfang getätigt werden können.

Dokumenttyp erfragen Den Dokumenttyp eines Modells kann man mit `EMFModelAccess.getDocumentType()` abfragen. Als Parameter ist entweder ein Modell (`Resource`) oder ein Modellelement `EObject`.

Parent Den Elternknoten eines EObject fragt man ab mit `EObject.eContainer()`.

Kinder Alle Kindknoten eines EObject werden abgefragt mit `EObject.eContents()`

Kindknoten, die über einen bestimmten Kantentyp verbunden sind, kann man mit `EMFModelAccess.getChildren(EObject eObject, EReference type)` abfragen.

Kindknoten, die von einem bestimmten Typ (Metaklasse) sind, kann man mit `EMFModelAccess.getChildren(EObject eObject, EClass type)` abfragen.

Eine Liste aller Typen (Metaklassen) von (vorhandenen) Kindern kann man mit `EMFModelAccess.getChildrenTypes(EObject eObject)` abfragen. (Achtung: Diese Abfrage bezieht sich auf eine Modellinstanz, nicht auf das Metamodell!)

Bäume Die Prüfung, ob ein `EObject` oder eine `Resource` einen Baum repräsentieren, erfolgt mit `EMFModelAccess.isTree(...)`. Es wird erwartet, dass Resourcen Bäume sind, wenn sie nur ein Wurzelelement enthalten, welches selbst wiederum einen Baum bildet.

Mit `EMFModelAccess.getTreeRoot(EObject eObject)` wird nach der Wurzel des Baumes gesucht, in dem sich das gegebene Object befindet.

Die Anzahl der Elemente in einem Baum kann mittels `EMFUtil.getEAllContentsSize()` abgefragt werden. Als Parameter ist entweder ein Modell (`Resource`) oder ein Modellelement `EObject`.

Mit `EMFModelAccess.traverse(EObject eObject, TreeVisitor visitor)` kann man einen Baum traversieren. `root` gibt dabei die Wurzel an. Der `TreeVisitor` stellt ein Interface zum Traversieren dar.

```
public interface TreeVisitor {
  public boolean preExecute(EObject object);
  public void postExecute(EObject object);
  public void init(EObject root) throws NoValidTreeException;
  public void finish(EObject root);
}
```

`init()` und `finish()` werden jeweils vor und nach dem Traversieren des Baumes mit dem Wurzelknoten als Parameter aufgerufen. Hier können (De-)Initialisierungen stattfinden. `preExecute()` wird in Reihenfolge einer Tiefensuche für jeden Knoten aufgerufen. Der Rückgabewert bestimmt, ob in den Teilbaum abgestiegen werden soll oder nicht. `postExecute()` wird nach durchlaufen des Teilbaums aufgerufen.

Mit `TreeVisitorImpl` steht eine leere Standardimplementierung des `TreeVisitor` zur Verfügung.

Referenzierte Elemente Benachbarte Elemente, die keine Kinder sind, bezeichnen wir als referenzierte Objekte. Für diese gibt es folgende Zugriffsmethoden:

`EMFModelAccess.getReferencedObjects(EObject eObject, EClass type)` gibt alle benachbarten Elemente vom Typ `type` zurück.

`EMFModelAccess.getReferencedObjects(EObject eObject, EdgeSemantic semantic)` gibt alle benachbarten Elemente zurück, die über eine Kante mit entsprechender Semantik (`semantic`) erreicht werden.

`EMFModelAccess.getReferencedObjects(EObject eObject, EdgeSemantic semantic, EClass type)` gibt alle benachbarten Elemente vom Typ `type` zurück, die über eine Kante mit entsprechender Semantik (`semantic`) erreicht werden.

Hinweis: Die Zugriffsmethoden für referenzierte Elemente schließen Eltern und Kinder grundsätzlich aus!

Geschwister Die Geschwister eines Elements sind die Elemente, die sich im gleichen Container befinden.

Mit `EMFModelAccess.getSiblings(EObject eObject)` lassen sich die Geschwister abfragen.

`EMFModelAccess.getLeftSibling(EObject eObject)` und `EMFModelAccess.getRightSibling(EObject eObject)` geben jeweils das linke bzw. rechte Geschwisterelement zurück. Ist ein Element das erste bzw. letzte Element in einem Container, wird `null` zurückgegeben.

Nachbarelemente Benachbarte Elemente sind all diejenigen Elemente, die über irgendeine Kante (Referenz oder Container/Containment) verbunden sind.

Mit `EMFModelAccess.getNodeNeighbors(EObject object)` werden alle benachbarten Element abgefragt.

`EMFModelAccess.getNodeNeighbors(EObject object, EReference... types)` gibt nur solche Nachbarn zurück, die über einen der gegeben Referenztypen (`types`) erreichbar sind.

`EMFModelAccess.getNodeNeighbors(EObject object, EClass... types)` gibt nur solche Nachbarn zurück, die einem der gegeben Typen (Metaklassen) (`types`) entsprechen.

`EMFModelAccess.getNodeNeighbors(EObject object, EdgeSemantic semantic)` gibt nur solche Nachbarn zurück, die über eine Referenz mit der gegeben Semantic (`semantic`) erreichbar sind.

`EMFModelAccess.getNodeNeighbors(EObject object, EdgeSemantic semantic, EClass... types)` gibt nur solche Nachbarn zurück, die über eine Referenz mit der gegeben Semantic (`semantic`) erreichbar sind und die einem der gegeben Typen (Metaklassen) (`types`) entsprechen.

5.5.3 EMFPath

Oftmals werden nicht direkt benachbarte Elemente benötigt, sondern entfernte. Hierzu gibt es Zugriffsmethoden, die Pfade benutzen:

Pfade können als String definiert werden. Der String enthält dann eine Liste von Referenznamen, die durch "/" getrennt werden. **Hinweis:** Derzeit können Pfade nur konkrete Referenznamen enthalten. Navigationselemente wie ".." werden noch nicht untersützt. Programmatisch werden Pfade durch `EMFPath`-Objekte repräsentiert. Diese können mittels `EMFMetaAccess.translatePath(EClass eClass, String path)` aus einem String erzeugt werden.

Abfragen mit Pfaden `EMFModelAccess.computeTargets(EObject start, EMFPath path)` gibt alle Knoten zurück, die über den angegeben Pfad erreicht werden.

`EMFModelAccess.computeTargetsWithoutBackStepping(EObject start, EMFPath path)` gibt alle Knoten zurück, die über den angegeben Pfad erreicht werden. Hierbei wird sichergestellt, dass beim Entlanglaufen des Pfades keine direkten Rückwärtsschritte erfolgen.

`EMFModelAccess.getElementPaths(EObject start, EMFPath path)` liefert eine Menge von Elementlisten, die entlang des Pfades liegen. Dabei entspricht eine Elementliste (`List<EObject>`) genau einem möglichen Weg entlang des gegeben Pfades. Alle möglichen Wege werden zurückgegeben (`List<List<EObject>>`).

`EMFModelAccess.getRemoteAttributeValue(EObject object, EMFPath path, EAttribute attribute)` kann einen Attributwert auf einem entfernten Objekt abfragen.