Karlsruhe Reports in Informatics 2014,1
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

ModelJoin

A Textual Domain-Specific Language for the Combination of Heterogeneous Models

Erik Burger, Jörg Henß, Steffen Kruse, Martin Küster,
Andreas Rentschler, Lucia Happe

2014

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Karlsruhe Institute of Technology

ModelJoin

A Textual Domain-Specific Language for the Combination of Heterogeneous Models

Erik Burger, Jörg Henß, Steffen Kruse, Martin Küster, Andreas Rentschler, Lucia Happe

January 21, 2014

Karlsruher Institut für Technologie
Fakultät für Informatik
Bibliothek
Postfach 6980
76128 Karlsruhe

Contents

List of Figures

1 Introduction

1.1 Motivation

In model-driven development, information is often spread across instances of multiple metamodels. These metamodels represent different aspects of the domain of interest, so it is possible that several instances of different metamodels, called *heterogeneous instances* in the following, represent a single entity in the domain of interest. If an MDD developer wants to make these semantic correspondences explicit, and integrate the information from heterogeneous instances into an integrated model, she or he has the following choices:

1. Modify the existing metamodels and instances, e.g. by introducing references between the metamodels.
2. Create a new metamodel that combines the desired information and migrate the instances.
3. Establish a mapping by a linking or a "glue" metamodel.

All these mentioned alternatives have serious drawbacks. Changing existing metamodels (1.) means that instances will have to be migrated and existing tooling has to be adapted. Furthermore, the semantic correspondences between the metamodels have to be determined and expressed by the new modeling concepts that are introduced during the modification of the metamodels. Creating a new metamodel (2.) also requires the definition of model transformations or migration scripts that convert existing instances; furthermore, it leads to a duplication of information in the existing models and the new models. Replacing the existing metamodels with a new, integrated metamodel is also not advisable since it would break the compatibility to existing tools and instances. Introducing a mapping or "glue" metamodel (3.) leaves the original metamodels unmodified, but also requires that the semantic correspondences are expressed as instances of this new model. If the instances are modified, the mappings have to be co-evolved, which can either be done manually, which causes additional effort, or automatically by model transformations, which are also hard to maintain.

To address these problems, we have created the *ModelJoin* approach, which automates the process of integrating information from heterogeneous metamodels and displays them as instances of a custom metamodel. To avoid the shortcomings of the approaches (1.)–(3.) mentioned above, *ModelJoin* was developed with the following requirements:

- The approach is non-invasive, i.e., existing metamodels and instances do not have to be modified.
- No model-transformations and metamodels have to be specified manually.

- The information need can be specified declaratively in a textual concrete syntax (TCS) query language.
- Queries can be modified and executed rapidly.

ModelJoin has been implemented prototypically using the modeling tools of the *Eclipse Modeling Framework (EMF)*[1], Xtext, Xtend, and QVT-O. The prototype offers a textual editor with syntax highlighting and content assist. It is available publicly and can be installed using the setup guide in the SDQ Wiki (see next section).

1.2 ModelJoin Resources

Since ModelJoin is an open source project, you can access the sources by checking out from the SVN repository. For issue tracking, a JIRA project exists which is also publicly accessible. There is also a page in the public SDQ wiki where details about the installation of ModelJoin can be found. See Table 1.1 for the URLs.

1.3 Structure of this document

This technical report is structured as follows: In chapter 2, we motivate the ModelJoin approach using a simple example. The ModelJoin DSL and its textual syntax are described in chapter 3, followed by chapter 4, which contains the documentation of the ModelJoin implementation prototype. Related work in the MDSD area is mentioned in chapter 5. The report concludes with a discussion of limitations and future work in chapter 6.

[1] http://www.eclipse.org/modeling/emf/

Wiki	http://sdqweb.ipd.kit.edu/wiki/ModelJoin
SVN	https://svnserver.informatik.kit.edu/i43/svn/code/MDSD/
	user: anonymous / password: anonymous
JIRA	http://www.palladio-simulator.com/jira/browse/MJ

Table 1.1: ModelJoin Resources

2 Example

In this chapter, we will motivate the ModelJoin approach with a simple motivating example. It is based on the library metamodel[1] commonly used as an example metamodel in Eclipse. This metamodel represents a simplified library, basically consisting of items and persons. Its purpose is both the management of circulation and the cataloging of library stock items. Typical library stock items are books and videocassettes. The metamodel is depicted in Figure 2.1.

As a complementary model source, we created a metamodel inspired by the well known Internet Movie Database (IMDb)[2] that is shown in Figure 2.2. Its purpose is to collect information about movies like title, year, and the cast. Furthermore, users of the IMDb can vote a numerical score for each movie.

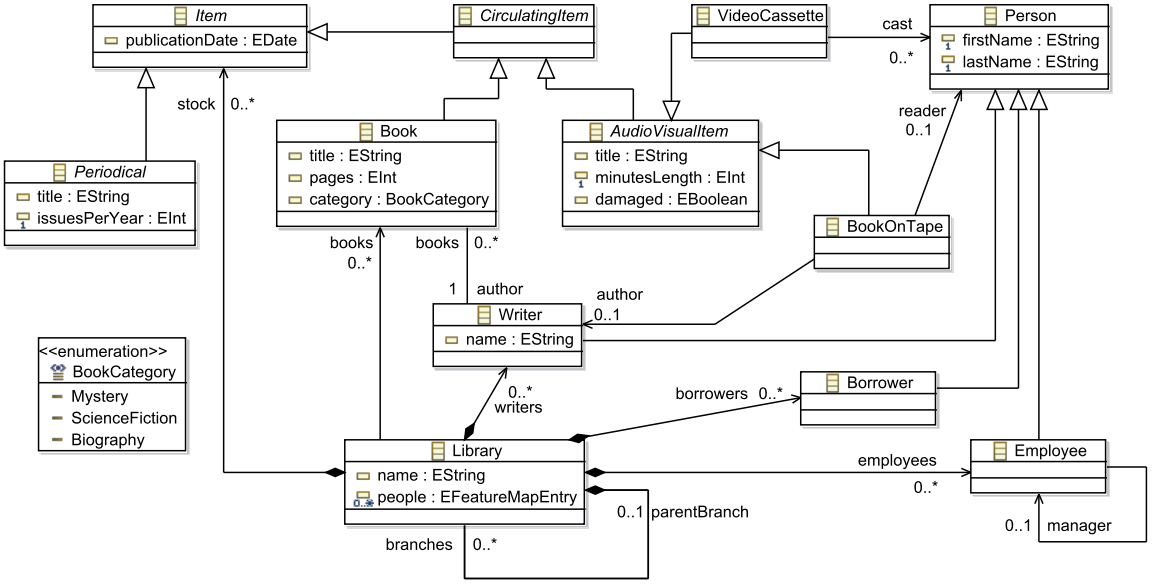

Figure 2.1: Metamodel of the Eclipse Library

We think that this is a typical situation: Metamodels taking two different perspectives have an overlap and represent the same entities. The information stored in different models might be

[1] http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.emf/org.eclipse.emf/examples/org.eclipse.emf.examples.library/?root=Modeling_Project

[2] http://www.imdb.com

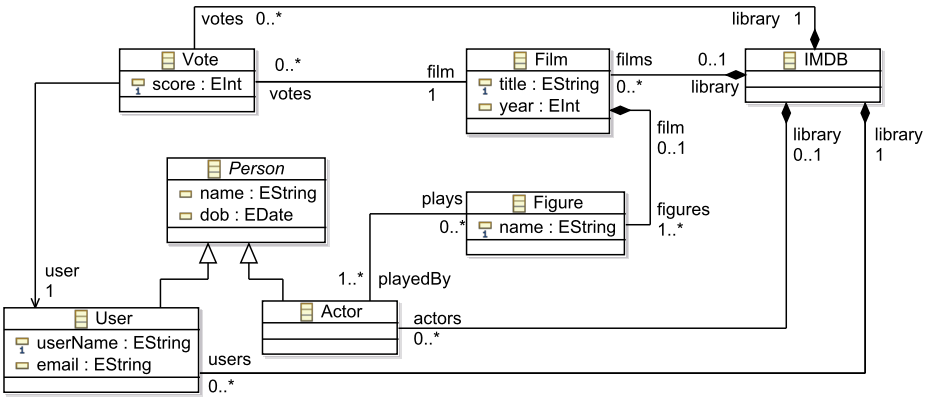

Figure 2.2: Metamodel of the Movie Database

useful in combination, but cannot be combined easily. State-of-the-art approaches require the writing of a transformation from the two metamodels to a (newly designed) third metamodel, which demands a lot of knowledge and effort. Integrating the metamodels using a decorator approach is an alternative way, but also requires a lot of effort.

In our example scenarios, users might also want to combine the information represented in both systems in a unified view. The classes Film and VideoCassette are good candidates for joined instances as they describe related entities and have a common attribute. Furthermore, the different kinds of persons and their roles can be used for information integration. The selection of instances based on attributes of writers or casts are typical use cases for queries to both theses models.

```
1 natural join imdb.Film with library.VideoCassette as jointarget.Movie {
2       keep attributes imdb.Film.year
3       keep attributes library.AudioVisualItem.minutesLength
4 }
```

Listing 1: Simple natural join (import statements omitted)

The first query resulting from both metamodels, seen in Listing 1, is joining the Film and VideoCassette classes on the common attribute title. As a speciality in this query, the attribute minutesLength is pushed down from the supertype AudioVisualItem representing a kind of metamodel refactoring possible in the ModelJoin language. Furthermore, the attribute year is retained from the class Film. The resulting metamodel should only contain one class with the attributes title, year, and minutesLength, as seen in

On the instance level, the result model will contain elements of type jointarget.VideoCassette whose attribute year comes from the IMDb and whose attribute minutesLength comes from the library. Only those elements will be present that have both an entity in IMDb and in the library with the same title.

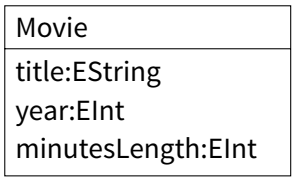

Figure 2.3: Joined metamodel for query 1.

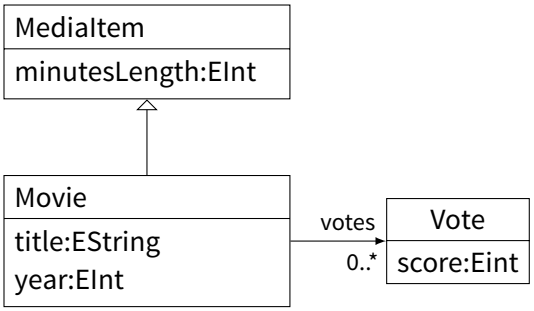

Figure 2.4: Joined metamodel for query 2.

```
1  theta join imdb.Film with library.VideoCassette as jointarget.Movie
2  where library.Videocassette.cast->forAll (p | imdb.Film.figures->playedBy->exists (a | p.
       firstname.concat("␣") .concat(p.lastName) == a.name)) {
3        keep attributes imdb.Film.year
4        keep outgoing imdb.Film.votes as type jointarget.Vote {
5        keep attributes imdb.Vote.score
6     }
7        keep supertype library.AudioVisualItem as type jointarget.MediaItem {
8           keep attributes library.AudioVisualItem.minutesLength
9     }
10 }
```

Listing 2: Theta join with condition (import statements omitted)

The second query (Listing 2) is a more complete version of the first query. In addition to the first query, references to the vote class and the supertype AudioVisualItem are maintained. Furthermore, the attributes score and minutesLength are kept for these additional classes. For the class AudioVisualItem a renaming to MediaItem is performed and the supertype relation between AudioVisualItem and VideoCassette is preserved. The resulting metamodel is shown in Figure 2.4.

The *where*-clause works as a filter on the instance level. It is expressed as an OCL query whose context is given by the joined elements. In the example, it selects those videocassettes whose casts have a corresponding actor (with the same name). Note that we need a concatenation of first and last names since Persons have a single attribute name in the IMDb metamodel, but a firstName and a lastName in the library metamodel. We have chosen a simple example for the sake of brevity here; of course, the conditions can have arbitrary complexity and are only limited by the expressive power of OCL.

3 The ModelJoin DSL

In this chapter, we will present the domain specific language (DSL) of ModelJoin.

3.1 Concept

The ModelJoin DSL has been created with the purpose of being similar to the Structured Query Language (SQL) of relational databases. We exploit several analogies to relational databases and relational algebra to make the semantics of ModelJoin better understandable. We assume that most users are familiar with the concept of queries in relational databases. The core concepts of relational databases such as database schema, tables, can be roughly mapped to concepts of MOF, such as classes, objects, attributes and so on. We will use the terminology of Ecore since the technical realization of ModelJoin is based on Ecore. Although Ecore is not a 100% implementation of MOF and uses a slightly different terminology, it is a widely used standard and can be seen as a reference implementation of MOF.

The analogies to relational databases which are used in our approach can be seen in Table 3.1: A model conforms to a metamodel in a similar way that tables conform to database schemata. A tuple can be compared to an object, since several tuples form a table in the same way that several objects form a model. Features of an object such as attributes or references correspond to columns of a relational table. Finally, a query on a database is similar to a model transformation. While in SQL, the statements in the query directly influence the database schema of the result, model transformations usually require a pre-defined target metamodel.

Existing query languages for Ecore such as EMFQuery[1] also offer a textual syntax that is inspired by SQL. They are however limited in the sense that the result set of such a query

[1] http://www.eclipse.org/modeling/emf/?project=query, http://wiki.eclipse.org/EMF%5C_Query2Home

relational concept	Ecore concept
database schema	metamodel
table	model
table row (tuple)	object/instance
column	feature (attribute/reference)
query	model transformation

Table 3.1: Analogy between relational concepts and MDD concepts

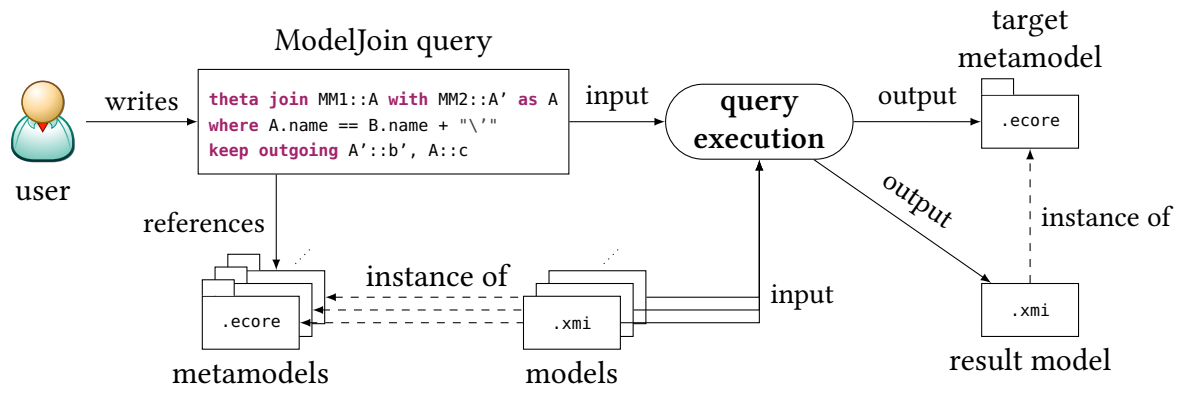

Figure 3.1: The workflow of models and metamodels in ModelJoin (from [9])

can only contain a subset of the set of instances that the query operates on, i.e., they are only *selectional*. It is not possible to create *projectional* queries that only retrieve specific properties of the underlying instances. More specifically, these query languages are lacking the concept of *joins* from relational algebra. This may be due to the fact that projectional queries on models require the definition of additional metamodels: When a SQL query containing joins is executed, the table schema of the result is determined by the `select`-statements in the query. Since the Ecore equivalent of a schema is a metamodel, according to Table 3.1, the metamodel of the result set of a projectional query is defined by the query itself. This means that the result set of a query execution contains instances of newly created metamodel, and the execution of a query would include a model-to-model transformation. Thus, a query language on models that supports projectional operators has to define the target metamodel and the properties of the instances of the result set.

In the prototypical implementation of ModelJoin as described in chapter 4, a new target metamodel is created upon every execution of the ModelJoin expression. The principle can be seen in Figure 3.1. This way, it is guaranteed that the model-to-model transformation, which is also generated, conforms to the target metamodel and produces the right result set. Theoretically, the result set of a ModelJoin execution could also consist of instances of a predefined metamodel, if this metamodel contains all the necessary elements that are determined by the parts of the ModelJoin expression. (This problem can be generalized to a compatibility relation between metamodels, using a change classification for metamodel evolution [5, 11]).

The intent behind ModelJoin was, however, not to develop yet another query language which offers only selectional operators. Instead, ModelJoin can be used to combine information from heterogeneous instances and only contain desired information, as defined by the user of ModelJoin. ModelJoin can also be used to define user-specific views on the underlying models, just like SQL queries can be persisted as views in a database. This mechanism can be applied in view-centric development approaches [9] that rely on a definition language for views on model-based data.

ModelJoin is also not a full-fledged transformation language. It cannot be used for arbitrary

kinds of model-to-model transformations, since its operations limit it to the projectional combination of models.

3.2 Abstract Syntax

A ModelJoin expression consists of *join-*, *where-*, and *keep-expressions*.

3.2.1 Join

The *Join* is the central concept of the ModelJoin DSL. The primary intention for joins is the merging of classes from different metamodels, which represent the same concept or have semantic overlaps. The join can, however also be used for analysis of elements from one metamodel, similar to joins in relational algebra. It is also possible to join a class with itself.

3.2.1.1 Natural Join

If two classes are joined with a natural join, common attributes of both source classes that are join-compatible are used as the join condition. Two attributes fulfil join-compatibility if they are of equal name and type, and have the same cardinalities. For each of these attribute pairs, an attribute of the same name and type has to exist in the target class.

For all instance pairs of the source classes that have equal values in these join-compatible attributes, a instance exists in the result set of the join.

In contrast to SQL, the natural join does not degenerate to the Cartesian product if no common attributes are found. Instead, no elements are added to the result set, and no requirements on the target class apply. A further difference to SQL is behaviour regarding the attributes in the target class. No attributes except the join-compatible attributes are added to the target metamodel automatically; if desired, this has to be defined by a `keep attributes` statement (see subsubsection 3.2.2.1).

Example We will use a simple example based on the motivating scenario of chapter 2, where video cassettes from a library and film items from the IMDb are joined into a Movie element. The metamodel of the result set is displayed in Figure 2.3 on page 9. The interesting lines of the query and the result set are displayed in Figure 3.2: In the left column, there are three instances of the class Film in the IMDB metamodel; in the right, there are two intances of the library class VideoCassette. These two classes share the join-compatible attribute title, which is also present in the join target class Movie.

In the result set, instances are created if there are instances in both source sets (IMDB and library). In the example, this is the case for the elements st1 and st8.

3.2.1.2 Outer Join

The layout of the target class for the outer join is similar to that of the natural join: common join-compatible attributes must exist in target class. On the instance level, a new instance is

Query header, example with Natural Join:

```
natural join imdb.Film with library.VideoCassette as jointarget.Movie {
```

Query header, example with Left Outer Join:

```
left outer join imdb.Film with library.VideoCassette as jointarget.Movie
```

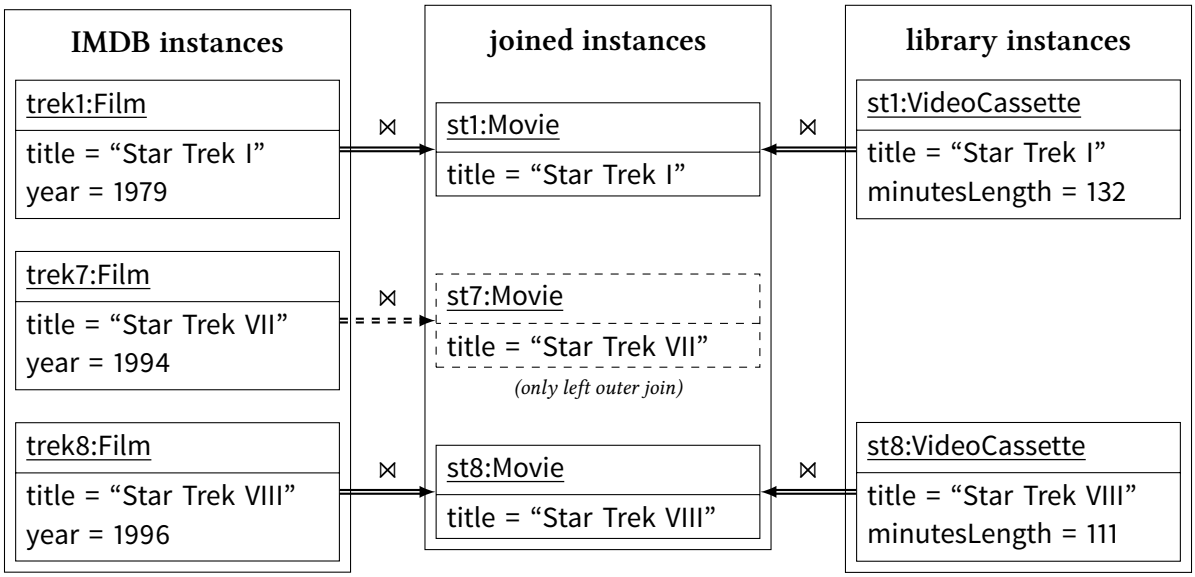

Figure 3.2: Result set after a natural join and left outer join

created for each instance in the respective source models, i.e., the first model in case of a left outer join, and the second model in case of a right outer join. The instance is created regardless of a matching instance in the other source model.

In the example in Figure 3.2, this is the case for the Film instance st8: A Movie element of the same name is created although there is no corresponding element in the library.

3.2.1.3 Theta Join

The θ-join offers the usage of arbitrary conditions for the join. The execution creates the Cartesian product of the input instances and then filters the set with the `where`-condition of the theta join. In the current implementation of ModelJoin, *OCL* can be used as a constraint language, since the QVT-O engine is used to execute the transformation.

The natural join can be seen as a special case of the theta join. The theta join does not respect common join-compatible attributes, so no attributes are added to the target class by default. If desired, this has to be made explicit by a *keep attributes* statement.

```
1 theta join imdb.Film with library.VideoCassette as jointarget.Movie
2 where library.Videocassette.cast->forAll (p | imdb.Film.figures->playedBy->exists (a | p.
      firstname.concat("␣") .concat(p.lastName) == a.name)) {
3       keep attributes imdb.Film.year
4       keep outgoing library.Videocassette.cast as type jointarget.Person {
5             keep attributes library.Person.firstName
6             keep attributes library.Person.lastName
7    }
8 }
```

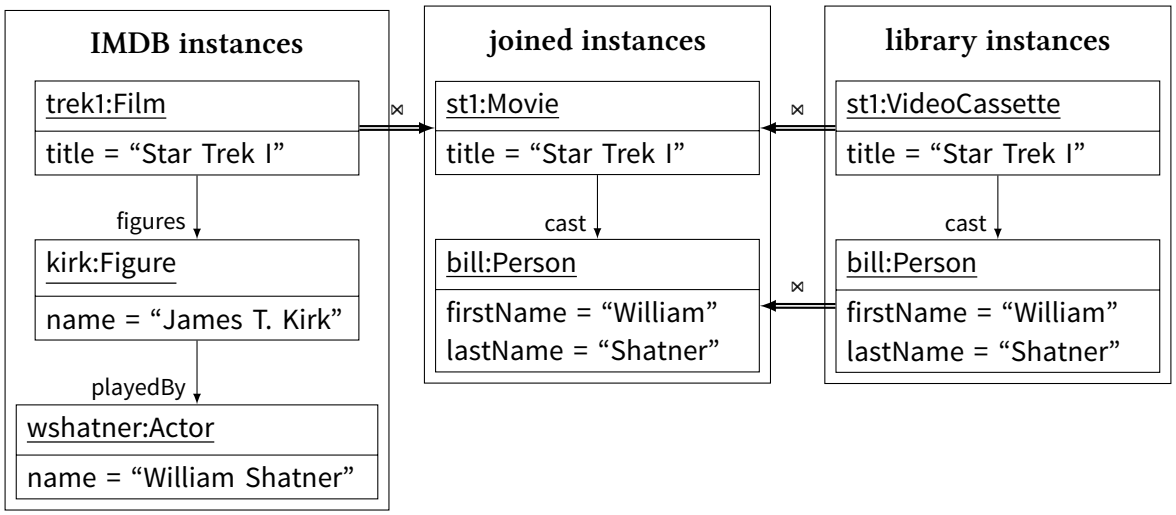

Figure 3.3: Example for the Theta Join

Example We extend the example of Listing 2 to illustrate the semantics of the Theta join. In Figure 3.3, the elements trek1:Film and st1:Videocassette are joined under the condition that all persons in the cast of a videocassette have a corresponding element in the IMDb instance (e.g., to distinguish several versions of a film).

3.2.2 Keep Statements

The keep expressions can be compared to the `select` statements of SQL since they define which features of the input objects should be contained in the target objects. Keep statements are nested inside join expressions or other keep expressions. In the concrete syntax of ModelJoin, we have made the nesting explicit with curly braces.

Since an `outer join` may have created objects in the target metamodel, it is always possible that no value for a feature exists, if the feature is contained in the “right” source class. Thus, all features in the target metamodel must have a lower cardinality of zero, so that valid instances of the target class that participates in an outer join can be created.

```
left outer join imdb.Film with library.VideoCassette as jointarget.Movie {
      keep attributes imdb.film.year
      keep attributes library.film.minutesLength
}
```

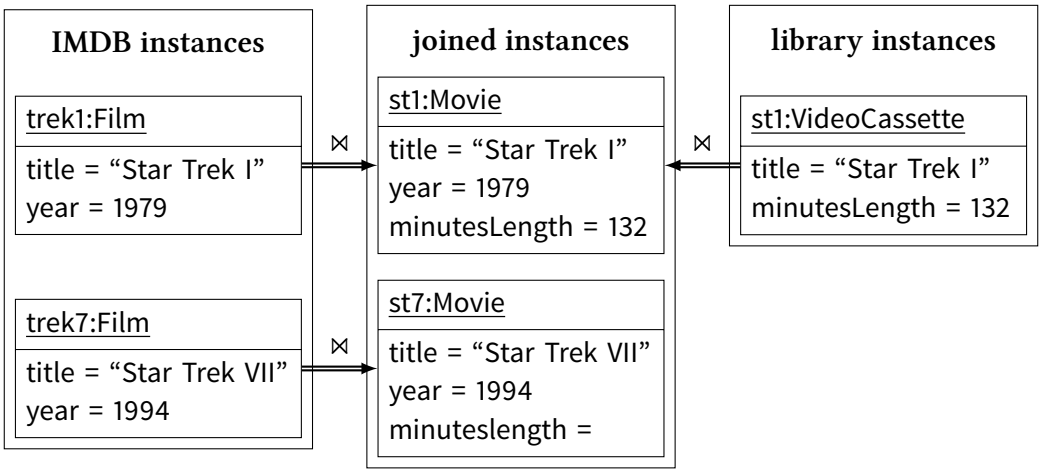

Figure 3.4: Result set after left outer join and keep attributes

3.2.2.1 Keep attributes

The `keep attributes` statement determines which attributes of either a left or right source class are set in the target instances. These attributes are required to exist in the target metamodel.

Since the `keep attributes` can be invoked on elements that participate in a `left outer join`, it is necessary to set the lower bound of all attributes in the target metamodel to 0. An example can be seen in Figure 3.4: The element st7:Movie was created by a left outer join. Since there is no matching element in the library instances, the attribute minutesLength cannot be set. To avoid having to set the element to a default or *null* value, the unsettable feature of EMF can be used, since it explicitly models the fact that a value has not been set (which is different from setting the value to *null*).

3.2.2.2 Aggregations

ModelJoin supports aggregation of attributes, similar to the aggregate functions in SQL with the `GROUP BY` statement. Currently, five aggregation types for numerical attributes are supported: `sum()`, `avg()`, `min()`, `max()`, and `size()`.

The aggregation in ModelJoin groups the elements by a certain reference. In the example, the votes for films are represented in the IMDB metamodel as single elements. With the aggregate operator, a new attribute averageRating can be introduced that gathers the average rating for a film.

```
natural join imdb.Film with library.VideoCassette as jointarget.Movie {
      keep aggregate avg(score) over Film.votes as Movie.averageRating
}
```

3.2.2.3 Calculated attributes

Calculated attributes are attributes whose value is computed in some form from other properties of the source models. In the prototypical implementation, they are specified using OCL.

The calculate attributes operator is the most generic way of specifying attributes in the target metamodel. The keep attribute operator and the aggregation operator can be simulated by a calculated attribute with an equivalent OCL expression.

Calculated attributes are not the same as *derived attributes* in EMF: The value of a derived attribute in a target model depends on other values in the *target models*, is not persisted (transient) and computed every time the attribute is accessed (volatile). Calculated attributes, in contrast, depend on values in the *source models* and are computed during the execution of the model transformation. In the target models, these attributes are non-transient and non-volatile.

Example In the following example, the calculated attribute topratings shows the number of users that have given a movie the highest rating (10 points).

```
natural join imdb.Film with library.VideoCassette as jointarget.Movie {
      keep calculated attribute imdb.Film.votes->select(v|v.score==10)->size()
      as Movie.topratings:EInt
}
```

3.2.2.4 Keep outgoing/incoming reference

The keep outgoing/incoming reference statement is used to include linked instances in the result set. This requires that the type/source of the reference also has to be mapped to an element in the target metamodel so that the instances can be created respectively.

In the example of Figure 3.3, the keep outgoing statement is used to include the reference cast in the target metamodel. Since the class Person is not created by any other statement in the ModelJoin expression, the class is created in the target metamodel during the execution of the keep outgoing statement.

3.2.2.5 Keep supertype

The keep supertype statements are used to define an inheritance hierarchy in the target metamodel. With a supertype statement, the supertype relation of a class that is joined or created through keep outgoing/incoming statement and one of its superclasses can be added in the target metamodel. If the class does not exist yet, the keep supertype statement will create it.

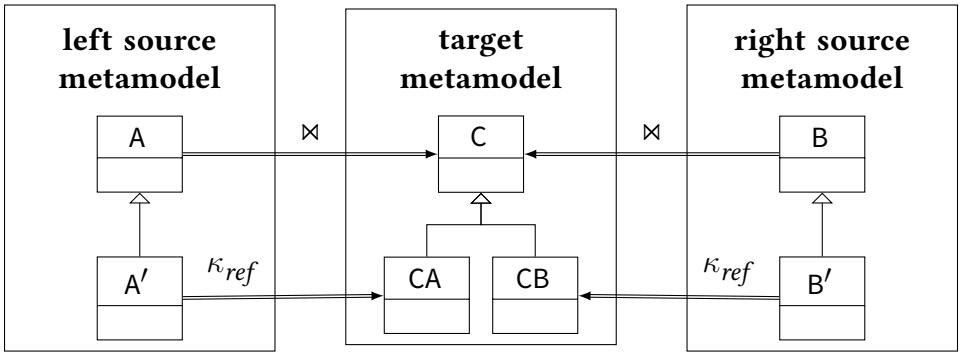

Figure 3.5: Example for keep subtype

In order to keep the automatisms in ModelJoin small, features such as attributes and references are not moved to a superclass automatically if they were contained in the superclass in the original model. The elements that are contained in the target superclass have to be made explicit with appropriate keep attribute/reference statements.

In the example of Listing 2 on page 8, the supertype AudioVisualItem is kept and renamed to MediaItem. The attribute minutesLength has been added explicitly to the class by the keep attribute statement in line 8 of the query.

3.2.2.6 Keep subtype

If instances of a subtype of, e.g., the source class that participates in a natural join are mapped to instances of the target metamodel, the most special subclass in the target metamodel is instantiated.

The keep subtype operator cannot be invoked on classes that participate in a join statement, since this would lead to ambiguous results.

Example An ambiguous case can be seen in Figure 3.5: If the ModelJoin expression is executed on two join-compatible instances of A′ and B′ respectively, the class of the resulting element is unclear; it could be either CA or CB. To resolve this problem, the semantic of the keep subtype operator could be changed to either take the left or the right argument as type in ambiguous cases, which would, in our opinion, be counter-intuitive. One could also generate a new class CAB which inherits from both CA and CB, but this would lead to the generation of many additional classes if there are several keep subtype statements, because the Cartesian product of all subclasses would have to be generated. For these reasons, we have decided to allow the keep subtype operator only on not-joined classes.

```
natural join a.A with b.B as jointarget.C {
      keep subtype a.A' as jointarget.CA
      keep subtype b.B' as jointarget.CB
}
```

3.2.3 Renaming

The join and keep operators contain an `AS` clause for the definition of the name of the elements in the target metamodel. If the rename statement is omitted, the elements in the target metamodel are named after the respective element in the source metamodel. In case of a join, the name of the left element is chosen as the name for the target element.

4 Technical Realisation

4.1 Concrete Syntax

package	`edu.kit.ipd.sdq.mdsd.mj.xtext`
language	Xtext

The concrete syntax of ModelJoin has been implemented using Xtext. The syntax definition is depicted in Listing 3.

```
1  grammar edu.kit.ipd.sdq.mdsd.ModelJoin with org.eclipse.xtext.common.Terminals
2  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
3  generate modelJoin "http://www.kit.edu/ipd/sdq/mdsd/ModelJoin"
4
5  Grammar:
6     (imports+=Import)*
7     (target+=Target)
8     (joinExpr+=JoinExpr)*;
9
10 JoinExpr :
11    (NaturalJoinExpr | LeftOuterJoinExpr | ThetaJoinExpr) 'as' targetType=CpxID
12    ('{'
13    (keepAttributesExpr+=KeepAttributesExpr)?
14    (keepAggregatesExpr+=KeepAggregateExpr)?
15    keepExpr+=KeepExpr*
16    '}')?
17    ;
18
19 NaturalJoinExpr:
20    'natural' 'join' left=[ecore::EClass|CpxID] 'with' right=[ecore::EClass|CpxID]
21 ;
22
23 LeftOuterJoinExpr:
24    'left' 'outer' 'join' left=[ecore::EClass|CpxID] 'with' right=[ecore::EClass|CpxID]
25 ;
26
27 ThetaJoinExpr:
28    'theta' 'join' left=[ecore::EClass|CpxID] 'with' right=[ecore::EClass|CpxID] 'where'
          condition=STRING
```

```
29 ;
30
31 KeepExpr :
32       (KeepTypeExpr | KeepOutgoingExpr | KeepIncomingExpr)
33       ('{'
34       (keepAttributesExpr+=KeepAttributesExpr)?
35       (keepAggregatesExpr+=KeepAggregateExpr)?
36       keepExpr+=KeepExpr*
37       '}')?
38 ;
39
40 KeepTypeExpr :
41    KeepSuperTypeExpr | KeepSubTypeExpr
42 ;
43
44 KeepSuperTypeExpr :
45    'keep' 'supertype' superType=[ecore::EClass|CpxID]
46       ('as' 'type' targetSuperType=CpxID)?
47 ;
48
49 KeepSubTypeExpr :
50    'keep' 'subtype' subType=[ecore::EClass|CpxID]
51       ('as' 'type' targetSubType=CpxID)?
52 ;
53
54 KeepOutgoingExpr :
55    'keep' 'outgoing' outgoing=[ecore::EReference|CpxID]
56       ('as' 'type' targetOutgoing=CpxID ('as' 'reference' targetReference=CpxID)?)?
57 ;
58
59 KeepIncomingExpr :
60    'keep' 'incoming' incoming=[ecore::EReference|CpxID]
61       ('as' 'type' targetIncoming=CpxID ('as' 'reference' targetReference=CpxID)?)?
62 ;
63
64 KeepAttributesExpr :
65    'keep' 'attributes' attribute=[ecore::EAttribute|CpxID] (',' attributes+=[ecore::
          EAttribute|CpxID])*
66 ;
67
68 KeepAggregateExpr :
69    'keep' 'aggregate' aggregate+=KeepAggregate (','aggregate+=KeepAggregate)*
70 ;
71
72 KeepAggregate:
```

```
73    KeepNumericalAggregate | KeepCollectionAggregate
74 ;
75
76 KeepNumericalAggregate :
77    aggregateKind=NumericalAggregateKind'('value=[ecore::EAttribute|CpxID]')'
78       'over' context=[ecore::EReference|CpxID] 'as' targetAttribute=CpxID
79 ;
80
81 KeepCollectionAggregate :
82    aggregateKind=CollectionAggregateKind (
83    '('value=[ecore::EAttribute|CpxID]')' 'over' context=[ecore::EReference|CpxID]
84    | '('value=[ecore::EReference|CpxID]')'
85    ) 'as' targetAttribute=CpxID
86 ;
87
88 enum NumericalAggregateKind :
89    SUM='sum' | AVG='avg' | MIN='min' | MAX='max'
90 ;
91
92 enum CollectionAggregateKind :
93    SIZE='size'
94 ;
95
96 WhereExpr :
97    'true'
98 ;
99
100 Projection :
101       star='*'
102    |  id=ID
103    |  cId=CpxID
104 ;
105
106 Import:
107    'import' importURI=STRING
108 ;
109
110 Target:
111    'target' targetURI=STRING
112 ;
113
114 CpxID : ID ('.' ID)+;
115
116 PackageQualifiedID : ID ('::' ID)* '::' (CpxID|ID) ;
```

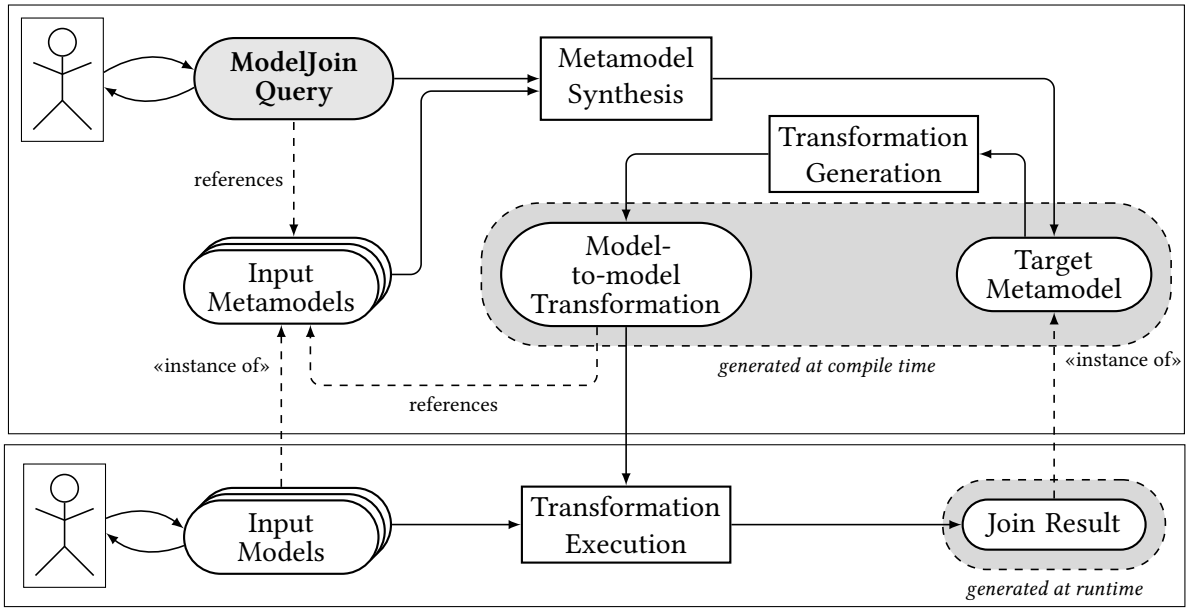

Figure 4.1: ModelJoin Workflow

Listing 3: ModelJoin Xtext Concrete Syntax

4.2 Workflow

In Figure 4.1, the workflow for the execution of a ModelJoin expression is depicted. The user writes a ModelJoin query that references input metamodels. Upon save, the workflows generate the target metamodel and the QVT transformation. To execute the ModelJoin query with actual instances of the input metamodels, the user has to launch an Eclipse *Run Configuration* and point it to the `.xmi` files that contain the input instances.

4.3 Generation of the target metamodel

package	`edu.kit.ipd.sdq.mdsd.mj.metamodel.generator`
language	Java

The metamodel generator is implemented in plain Java.

Since both the transformation generation and the metamodel generation need to know the relations from the input metamodels to the target metamodel, the ModelJoin query would have to be parsed twice to calculate the relation. Since this would lead to duplicate code and redundant parsing of the query, we decided to annotate the target metamodel with tracing information.

Structural Primitives	join	keep super-/subt.	keep reference	keep (calc.) att./aggr.
Create package	x	x	x	-
Create class	x	x	x	-
Create attribute	x	-	-	x
Create reference	-	-	x	-
Create data type	-	-	-	x
Create enum	-	-	-	x
Non-structural Primitives				
Add super type	-	x	-	-

Table 4.1: Operations used for the metamodel synthesis

4.3.1 Construction of the target metamodel

We use an operator-based approach for building the target metamodel from a ModelJoin query. Each statement of a ModelJoin query is therefore translated to a set of operations necessary to reflect its semantics.

We build upon the set of metamodel operations introduced by Hermansdörfer et al. [20]. Table 4.1 gives an overview on the subset of operations that are used for synthesizing a joined metamodel. For each operation, corresponding statements are marked by an x. Some operations are only executed when the source elements of a statement are of specific types.

The required operations are extracted from the query by recursively traversing the statement-tree. The extraction is described in Algorithm 1. Starting from the join statements the nested keep expressions are translated to metamodel operations. Each kind of operation is stored in a separate set.

As first step, all *Create Package* and *Create Class* operations are performed. Furthermore, data types, enums and literals are created.

As next step, the *Add Super Type* operations are performed to define a hierarchy of classes.

As last step, based on the hierarchy, the references and attributes are added. Starting from the topmost set of classes, it is checked which references and attributes can be added to the class. When an operation is encountered that is meant to create an attribute or reference already present in a superclass, it is discarded as it is subsumed by the superclass.

4.3.2 Annotations in the target metamodel

During the generation of the target metamodel, the metamodel generator extracts information from a ModelJoin expression such as which elements of the source metamodel were joined to an element of the target metamodel, which was the attribute for the join condition, etc. The information is stored in the target metamodel using `EAnnotation` elements which reference the elements in the source metamodels directly. Thus, the generated target metamodel will contain

name/source	reference	details
sourceModels	the source metamodels of the ModelJoin query	should be attached to the root package
createdBy	–	operator: the type of operator that caused the creation of the element (naturaljoin, thetajoin)
classTrace	the single source element	–
classTrace.left	the left source element if the operator is a join	–
classTrace.right	the right source element if the operator is a join	–
isJoinAttribute	an attribute that was part of the join condition; in case of several attributes, an annotation for each attribute is created	–
whereCondition	–	ocl: the OCL expression

Table 4.2: Annotation types in the generated metamodel

references to the source metamodels in its EAnnotation elements.

EAnnotations have a name (called source in EMF), an element that they refer to (reference), and can contain additional information details in key/value pairs. Since EAnnotations are not typed, we introduce naming conventions for the different annotation types (see Table 4.2).

4.4 Transformation Generation (QVT-O)

package	`edu.kit.ipd.sdq.mdsd.mj.transformation.generator`
language	Xtend2

4.4.1 Overview

The model-to-text transformation generation is implemented in Xtend2. It takes the annotated target metamodel as input and generates QVT-O text using Xtend's template mechanism. The transformation generation does not need the original Xtext query since all information that is necessery for generation can be determined from the annotations (see Table 4.2).

For each operation type, the Xtend2 template generates two sections in the QVT-O transformation: First, a section in the `main()` method in which the appropriate model elements are selected and connected by a `map` statement; second, the definition of the `mapping` itself.

Algorithm 1 Operation extraction algorithm

```
1: Set o := {}                                                    ▷ The set of operations

2: procedure processQuery
3:     for j : joins do
4:         processJoin(j)
5:     end for
6: end procedure

7: procedure processJoin(Join j)
8:     o ← CreatePackage(j.target.package))                       ▷ Create class operations
9:     o ← newCreateClass(j.target)
10:    o ← newCreateAnnotation("classtrace.left", j.target, j.left)   ▷ Create annotations
11:    o ← newCreateAnnotation("classtrace.right", j.target, j.right)
12:    o ← newCreateAnnotation("createdBy", j.type)

13:    features ← getJoinFeatures(j)                              ▷ Create joinFeatures
14:    for f : features do
15:        o ← newCreateType(f.type)
16:        o ← newCreateFeature(j.target, f.type, f.name)
17:        o ← newCreateAnnotation("joinattribute", f)
18:    end for
19:    processKeeps(j)
20: end procedure

21: procedure processKeeps(Expression e)
22:    for k : e.keeps do
23:        processKeep(k, e)
24:    end for
25: end procedure

26: procedure processKeep(KeepSupertypeExp/KeepSubtypeExp kse, Expression context)
27:    o ← newCreatePackage(kse.target.package)                   ▷ Create super/sub class operations
28:    o ← newCreateClass(kse.target)
29:    o ← newCreateAnnotation("classtrace", kse.target, kse.source)   ▷ Create annotations
30:    o ← newCreateAnnotation("createdBy", "superType"/"subtype")
31:    o ← newCreateSuperType/CreateSubType(kse.target, context.target)
32: end procedure

33: procedure processKeep(KeepReferenceExp kre, Expression context)
34:    o ← newCreatePackage(kre.target.package)                   ▷ Create super class operations
35:    o ← newCreateClass(kre.target)
36:    o ← newCreateAnnotation("classtrace", kre.target, kre.source)   ▷ Create annotations
37:    o ← newCreateAnnotation("createdBy", kre.type)
38:    if kre is KeepOutgoingExp then
39:        o ← newCreateFeature(context.target, kre.target, kre.name)
40:    else
41:        o ← newCreateFeature(kre.target, context.target, kre.name)
42:    end if
43: end procedure

44: procedure processKeep(KeepAttributesExp kae, Expression context)
45:    o ← newCreateType(kae.type)
46:    o ← newCreateFeature(context.target, kae.type, kae.name)
47: end procedure
```

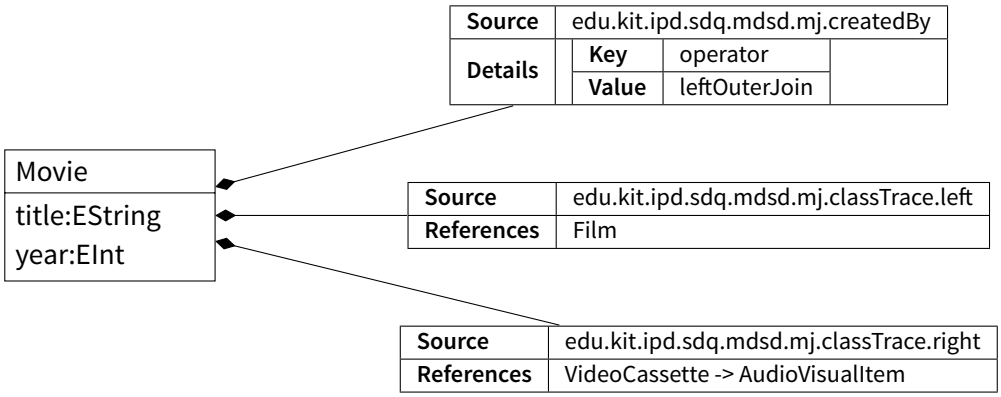

Figure 4.2: Example: A class in the generated metamodel with annotations

Algorithm 2 Metamodel generation algorithm

```
1:  procedure generateMetaModel

2:      for pacOp : o.filter(CreatePackage) do
3:          pacOp.perform;
4:      end for

5:      for classOp : o.filter(CreateClass) do
6:          classOp.perform
7:      end for

8:      for superTypeOp : o.filter(CreateSuperType) do
9:          superTypeOp.perform
10:     end for

11:     featureOps = o.filter(CreateFeature)                    ▷ Create features
12:     for i=0..targetModel.classDepth do
13:         classes = targetModel.classes.filter(level==i);
14:         for class : classes do
15:             for createFeatureOp : featureOps.filter(target==class) do
16:                 if !class.allSuper.features.contains(f) then
17:                     createFeatureOp.perform
18:                 end if
19:             end for
20:         end for
21:     end for

22:     for annotationOp : o.filter(CreateAnnotation) do        ▷ Create annotations
23:         if model.contains(annotationOp.target) then
24:             annotationOp.perform
25:         end if
26:     end for
27: end procedure
```

4.4.2 Generation of Code for the ModelJoin Operators

4.4.2.1 Natural and Outer Joins

The natural and outer joins are executed by forming the cartesian product of the source objects and then filtering the elements in the `when`-clause of the mapping statement. The `when`-clause realises the join-compatibility condition of subsubsection 3.2.1.1.

Since the mapping is invoked on the "left" elements of the natural join with the "right" elements as a parameter, the QVT mapping table only contains entries from the left elements to the target elements. In later operations, however, it is necessary to check for the relation of target elements to all source elements (including the "right" elements). For this reason, an additional helper mapping is generated which only sets the mapping from the a source element to the target element. The mapping, e.g. `naturalJoin_update_VideoCassette`, is invoked at the end of a natural join mapping.

4.4.2.2 Theta Join

The theta join works in quite a similar way as the natural and outer joins, except for the fact that the `when`-condition of the mapping is extracted from the annotation. In the theta join expression, OCL is used as the language in the `where`-condition. Like in SQL, the class names of source classes can be used in the OCL expression. During the transformation generation, the qualified names of the classes are replaced by the respective variables or `self`.

4.4.2.3 Keep outgoing/incoming reference

For the keep outgoing and incoming operators, a distinction must be made depending on whether the instance at the other end of the link has already been mapped an instance in the target model. If the instance has already been mapped, then the link is updated with the existing instance; otherwise, a new instance is created and linked. For each of these two cases, a mapping is generated in the QVT-O file, e.g. `update_keepOutgoing_cast` and `update_keepOutgoing_and_create_cast`.

4.4.2.4 Keep (calculated) attribute/aggregate

Keep attribute operators are translated into `inout` mappings which update existing instances of the target class by setting the attribute value to either the value of the source instance (keep attribute) or to an OCL expression (keep calculated attribute).

The mapping for aggregations is a special case of a calculated attribute with pre-defined OCL operations for the aggregation functions.

5 Related Work

Approaches related to ModelJoin in the area of model-driven software development can be classified according to the three central concerns as follows:

1. Approaches for the synthesis of metamodels
2. Approaches for the synthesis of models
3. Approaches for the creation and evaluation of queries on models

Since an important aspect of ModelJoin is the creation of a large number of artifacts on the fly, we further looked at whether each aspect is treated in a static or dynamic manner, depending on whether use of predefined or hard coded artifacts or implicit or generated artifacts is made.

A lot of work is done in the area of model synthesis to enable collaborative modeling, for example in the use of version control systems for models. The aim is to calculate the difference between versions of models and to merge models of different versions – both for MOF-based models ([1]) and for EMF, like the diff and merge algorithms of EMF Compare [8]. Another common task for model synthesis is the handling of metamodel evolution. Here, models are synthesized or "updated" to restore syntactic or semantic conformance [20, 13]. These approaches are commonly static in nature and (in contrast to ModelJoin) rely on the fact that the treated models are related – by either stemming from the same base model or conforming to different versions of the same metamodel.

The Epsilon Merging Language [23] supports the merging of models from different metamodels like ModelJoin. In contrast to ModelJoin, it requires the target metamodel to be created manually before merging rules can be defined. Thus the approach is static.

The VirtualEMF project [14] introduces virtual models as a run-time solution for adapting one model or potentially merging models from numerous sources. While the merged models are created on the fly and on demand at run-time, in contrast to ModelJoin, the merged metamodel and a weaving model have to be defined beforehand and are not generated. We see our approach to be complementary, as both artifacts could be generated using our approach.

The EMF Facet project [15] provides a mechanism to extend an existing metamodel and conforming models with new elements, without changing the original artifacts. The approach is thus related to both the synthesis of metamodels and models. In contrast to ModelJoin however, it does not integrate two different metamodels.

In general, the join operator represents in part a special form of model transformation and is (as it is declarative) especially related to declarative transformation languages like QVT Relations [26] and ATL [21]. Yet in contrast, it is specifically tailored to easily and quickly define views on two similar metamodels. This imposes restrictions (see **??**) but in turn no predefined target metamodel is required as is needed for general-purpose transformation languages. The

general purpose languages should be used for cases too complex for ModelJoin; here ModelJoin can still serve as a good starting point.

The EMF-IncQuery framework [17] tackles the problem of interconnecting heterogeneous models without setting hard links between their metamodels. Instead, incremental queries are executed to calculate derived features of EMF models. The approach also features a caching mechanism but is not completely non-intrusive, since the source metamodels have to be modified by adding the derived features.

The ModelJoin approach is related to the field of model composition [19, 6] and aspect-oriented modelling (AOM), which also include view-based modeling techniques [22]. Other approaches include Kompose [16], AMW [2] and GGT [7]. These tools merge arbitrary metamodels in a similar way to ModelJoin via a composition rule set, but require the definition of a linking or "glue" model between the source metamodels.

Other approaches for the management of heterogeneous models use a central, fixed metamodel as a hub; bidirectional transformations have to be specified for all metamodels that are to be supported: The OSM approach [3] has been implemented in KobrA [4]. More tool-driven approaches include ModelBus [18], which is focused on the interoperability of heterogeneous modeling tools, and DuALLy [25], which uses higher-order transformations based on ATL for architectural description languages.

6 Limitations and Future Work

6.1 Limitations of the language

6.1.1 Nesting of Joins

In the current version of ModelJoin, it is not possible to nest join operations. For example, the following query would select all actors that have appeared as figures in a film and, and join it with the users in the library:

```
natural join library.Person with
      {natural join imdb.Actor with imdb.Figure}
```

It is not possible to write this in one single query in ModelJoin at the moment. To acquire the desired result, the inner natural join would have to be executed separately; then, a query that uses the library metamodel and the generated target metamodel of the first query as source metamodels can be written. We plan to support this feature in upcoming versions of ModelJoin.

6.1.2 Joining over References

The natural and outer join operators are defined over attributes; it is currently impossible to join over references. A join over a reference would have the join condition that the same element is linked to the classes that should be joined. This is shown in an example in Figure 6.1: If class A and B are joined over the reference ref, it would require that there are identical instances linked to instances of A and B. Since A and B can be part of distinct source models, the classes at the end of ref may also be distinct, like the classes A' and B' in the example.

Thus, a join over a reference would require that the classes at the end of the references are either identical or have also been joined, so that the identity of the instances can be determined.

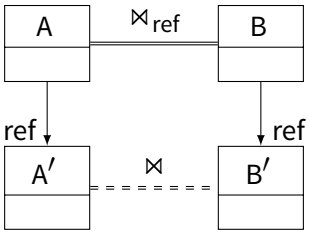

Figure 6.1: Example for joining over references

In future work, we plan to include a join operator for joining over references. In the execution algorithm, the reference-joins will be computed after other joins, since it is a precondition that the classes at the end of the reference are identical or have been joined.

6.2 Editability

The queries that are specified with ModelJoin are read-only. In the course of the Vitruvius research project [10, 24], an extension to ModelJoin is being developed that extends it by a facility for editing and synchronisation. In Vitruvius, ModelJoin is used for the rapid, on-the-fly definition of customized view types, in a similar way that SQL queries can be stored as views. If such a view type is to be persisted for future usage, the generated artefacts (such as metamodels, tracing information and transformations) can be used as a starting point for editable, synchronized views.

In a multi-metamodel scenario, which we aim to support with ModelJoin, editability of these views only makes sense if the instances of the source metamodels are also synchronized with each other, and not only the views with the metamodels. This is due to two reasons: First, we consider synchronization of views via a central model (or a set of models) superior to direct synchronization between views for reasons of complexity. Second, since we support view types with a multi-metamodel projectional scope, it is possible that, for example, one flexible view type affects instances of Metamodels M_1 and M_2, while another flexible view type affects instances of M_2 and M_3. To keep the central set of models consistent, it is necessary to have synchronization even for those instances that are not directly affected by a flexible view type.

6.3 Re-Use of Target Metamodels

A ModelJoin query always contains sufficient information to generate the target metamodel from it. In the current implementation of ModelJoin, the target metamodel is always generated during the execution of a query. Although this guarantees that the generated transformations always fit to the target metamodel, it has the disadvantage that the generated metamodels cannot be re-used for further purposes, such as further model transformations, specialized editors, or vizualizations of the models. Even in the special case that a metamodel is joined with itself, the target metamodel is not identical to the source metamodel, but depends on the structure of the ModelJoin query.

Since the target metamodel is dependent from the query, even small changes in the query lead to a new version of the target metamodel, although the target metamodel of the previous version of the query would be a valid metamodel for the result set of the current query. To determine whether the changes between two metamodels invalidate the instances of the previous version of the metamodel, we have created a conformance check [12], which is based on the metamodel evolution tool Edapt [20] and the rule-based engine DROOLS[1].

[1] http://www.jboss.org/drools/

The conformance check is not integrated in ModelJoin yet. To integrate it, the ModelJoin workflow (Figure 4.1) could be extended by a conformance check after the Metamodel Synthesis step. The check would have to be connected to a metamodel repository that stores the possible target metamodels. Upon generation of a target metamodel, the conformance checker decides whether an existing metamodel can be used as the target metamodel and passes this information to the transformation generator. The join result is then created as an instance of the metamodel from the repository rather than an instance of the newly generated metamodel.

Bibliography

[1] Marcus Alanen and Ivan Porres. "Difference and Union of Models". In: *"UML 2003" – The Unified Modeling Language, Modeling Languages and Applications 6th International Conference, San Francisco, CA, USA, October 20–24, 2003, Proceedings*. Ed. by Perdita Stevens, Jon Whittle, and Grady Booch. Vol. 2863. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer Verlag, 2003, pp. 2–17. ISBN: 978-3-540-20243-1.

[2] *Atlas Model Weaver*. URL: http://www.eclipse.org/gmt/amw/.

[3] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Vol. 69. Communications in Computer and Information Science. Berlin/Heidelberg: Springer, 2010, pp. 206–219. ISBN: 978-3-642-14819-4.

[4] Colin Atkinson et al. "Modeling Components and Component-Based Systems in KobrA". In: *The Common Component Modeling Example*. Ed. by Andreas Rausch et al. Vol. 5153. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, 2008, pp. 54–84. URL: http://dx.doi.org/10.1007/978-3-540-85289-6_4.

[5] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "Model-based Performance Prediction with the Palladio Component Model". In: *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, Feb. 2007.

[6] Jean Bézivin et al. "A Canonical Scheme for Model Composition". In: *Model Driven Architecture – Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Vol. 4066. LNCS. Springer Berlin / Heidelberg, 2006, pp. 346–360. ISBN: 978-3-540-35909-8. URL: http://dx.doi.org/10.1007/11787044_26.

[7] Bouzitouna, Salim and Gervais, Marie-Pierre and Blanc, Xavier. "Model Reuse in MDA". In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'05)*. Las Vegas, USA, June 2005.

[8] Cédric Brun and Alfonso Pierantonio. "Model Differences in the Eclipse Modelling Framework". In: *UPGRADE The European J for the Informatics Professional* IX.2 (2008), pp. 29–34. URL: http://www.cepis.org/upgrade/files/2008-II-pierantonio.pdf.

[9] Erik Burger. "Flexible views for rapid model-driven development". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '13. Montpellier, France: ACM, 2013, 1:1–1:5. ISBN: 978-1-4503-2070-2. URL: http://doi.acm.org/10.1145/2489861.2489863.

[10] Erik Burger. "Flexible Views for View-Based Model-Driven Development". In: *Proceedings of the 18th international doctoral symposium on Components and architecture*. WCOP '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 25–30. ISBN: 978-1-4503-2125-9. URL: http://doi.acm.org/10.1145/2465498.2465501.

[11] Erik Burger and Boris Gruschko. "A Change Metamodel for the Evolution of MOF-Based Metamodels". In: *Proceedings of Modellierung 2010*. Ed. by Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr. Vol. P-161. GI-LNI. Klagenfurt, Austria, Mar. 24–26, 2010. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/burger2010a.pdf.

[12] Erik Burger and Aleksandar Toshovski. "Difference-based Conformance Checking for Ecore Metamodels". In: *Proceedings of Modellierung 2014*. GI-LNI. To appear. Vienna, Austria, Mar. 19–21, 2014.

[13] Antonio Cicchetti et al. "Automating Co-evolution in Model-Driven Engineering". In: *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*. EDOC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 222–231. ISBN: 978-0-7695-3373-5. URL: http://dx.doi.org/10.1109/EDOC.2008.44.

[14] Cauê Clasen, Frédéric Jouault, and Jordi Cabot. "VirtualEMF: A Model Virtualization Tool". In: *Advances in Conceptual Modeling. Recent Developments and New Directions*. Ed. by Olga De Troyer et al. Vol. 6999. LNCS. Springer Berlin / Heidelberg, 2011, pp. 332–335. ISBN: 978-3-642-24573-2. URL: http://dx.doi.org/10.1007/978-3-642-24574-9_43.

[15] *EMF Facet*. URL: http://www.eclipse.org/facet/.

[16] Franck Fleurey et al. "A Generic Approach for Automatic Model Composition". In: *Models in Software Engineering*. Ed. by Holger Giese. Vol. 5002. LNCS. Springer Berlin / Heidelberg, 2008, pp. 7–15. ISBN: 978-3-540-69069-6. URL: http://dx.doi.org/10.1007/978-3-540-69073-3_2.

[17] Ábel Hegedüs et al. "Query-Driven Soft Interconnection of EMF Models". In: *Model Driven Engineering Languages and Systems*. Ed. by Robert France et al. Vol. 7590. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 134–150. ISBN: 978-3-642-33665-2. URL: http://dx.doi.org/10.1007/978-3-642-33666-9_10.

[18] Christian Hein, Tom Ritter, and Michael Wagner. "Model-Driven Tool Integration with ModelBus". In: *Workshop Future Trends of Model-Driven Development*. 2009.

[19] Christoph Herrmann et al. "An Algebraic View on the Semantics of Model Composition". In: *Model Driven Architecture- Foundations and Applications*. Ed. by David Akehurst, Régis Vogel, and Richard Paige. Vol. 4530. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 99–113. ISBN: 978-3-540-72900-6. URL: http://dx.doi.org/10.1007/978-3-540-72901-3_8.

[20] Markus Herrmannsdörfer, Sander D. Vermolen, and Guido Wachsmuth. "An extensive catalog of operators for the coupled evolution of metamodels and models". In: *Proceedings of the Third international conference on Software language engineering*. SLE'10. Berlin/Heidelberg: Springer, 2011, pp. 163–182. ISBN: 978-3-642-19439-9. URL: http://www4.in.tum.de/~herrmama/publications/SLE2010_herrmannsdoerfer_catalog_coupled_operators.pdf.

[21] F. Jouault and I. Kurtev. "Transforming models with ATL". In: *Satellite Events at the MoDELS 2005 Conference*. Vol. 3844. LNCS. Berlin: Springer Verlag, 2006, pp. 128–138. URL: http://doc.utwente.nl/61719/.

[22] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. "Aspect-oriented multi-view modeling". In: *Proceedings of the 8th ACM international conference on Aspect-oriented software development*. AOSD '09. Charlottesville, Virginia, USA: ACM, 2009, pp. 87–98. ISBN: 978-1-60558-442-3. URL: http://doi.acm.org/10.1145/1509239.1509252.

[23] Dimitrios Kolovos, Richard Paige, and Fiona Polack. "Merging Models with the Epsilon Merging Language (EML)". In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz et al. Vol. 4199. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 215–229. ISBN: 978-3-540-45772-5.

[24] Max E. Kramer, Erik Burger, and Michael Langhammer. "View-centric engineering with synchronized heterogeneous models". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '13. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2. URL: http://doi.acm.org/10.1145/2489861.2489864.

[25] I. Malavolta et al. *Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies*. Tech. rep. 1. Jan. 2010, pp. 119–140.

[26] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group. January 2011. URL: http://www.omg.org/spec/QVT/1.1/.