T U M

I N S T I T U T F Ü R I N F O R M A T I K

BOTL
The Bidirectional Object Oriented Transformation Language

Peter Braun, Frank Marschall

TUM-I0307
Mai 03

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-I0307-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

BOTL
The Bidirectional Object Oriented Transformation Language

Peter Braun and Frank Marschall

May, 2003

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching
Germany

Abstract

Modern software engineering bases heavily on models that represent an abstraction of a system under development. Object orientation has established as a standard modeling technique for the specification of software systems. Abstraction and refinement relations within the development process require the transformation of different models. Today there exists no language that is capable to unambiguously define transformations of object oriented models.

This report presents the Bidirectional Object oriented Transformation Language (BOTL). BOTL is based upon a precise, formal foundation and a comprehensible, graphical notation. Further techniques are presented that allow to verify whether a BOTL specification is applicable and whether it will produce models that are conform to a given metamodel. Another property that is introduced is the bijectivity of BOTL specifications.

A running example is used throughout the report to illustrate the BOTL concepts. Further an extension for inheritance and a mapping to the UML are presented as detailed applications of the BOTL.

BOTL is intended to be supported by a tool that uses the presented verification techniques and generates code for the transformation of object oriented models.

Contents

List of Figures

List of Tables

Definitions

Theorems

Lemmas

1 Motivation

In virtual every engineering discipline the key factor for successful development of any kind of products is the use of appropriate models. Thereby a model abstracts from reality by suppressing irrelevant details. According to the specific needs of a discipline different kinds of models are used. Whereas models in the area of construction are usually building plans, chemical engineers depend heavily on abstractions of chemical processes. In the discipline of software engineering object oriented models are widespread to describe software systems. Object orientation is used as general modeling technique to define the structure of data. Thus object oriented models are also used at a meta level to define other types of models as for the Unified Modeling Language (UML) [OMG02].

Refinement, abstraction, refactoring, and integration of models are special cases of model transformations that can be found in many areas of software engineering. Since object orientation is a rather young modeling technique there is still a lack of mature specification techniques for transformation of object oriented models.

This work introduces the Bidirectional Object oriented Transformation Language (BOTL) as a language for the specification of such transformations.

In the following we will sketch some typical application scenarios for model transformations and introduce the BOTL, which results from ongoing research in the projects AUTOMOTIVE and KOGITO.

1.1 Application Areas for Model Transformations

In this section we sketch three exemplary application areas where model transformations are necessary within software development. We are actively involved in these applications areas within the projects AUTOMOTIVE [aut03] and KOGITO [kog03]. Thus we are convinced that the use BOTL can be of great value for these, and of course other fields.

1.1.1 Model Driven Architecture (MDA)

In software engineering models are essential for the description of software systems under development and their environment at different layers of abstraction. The MDA [Sol00, ORM01]

introduced by the Object Management Group (OMG) outlines a model based software engineering approach that explicitly separates models at three different abstraction layers. The most abstract model layer, the Computational Independent Model (CIM), is used to specify the concepts of the application domain. This model layer is refined into a Platform Independent Model (PIM) that contains already computational information about the specified system but still is free of platform specific realization details. Platform specific details, like e.g. information about whether a PIM component is realized as a session Enterprise Java Bean (EJB) or a entity EJB [Mic03], are added in the Platform Specific Model (PSM). Thus the PSM again is a refinement of the PIM.

A developer obtains a refined model from an existing one by transforming it either by hand or with the help of an appropriate tool. The MDA identifies also possible mappings from a PIM to a PIM, from a PSM to a PSM, and from a PSM to a PIM. These mappings are refinement steps within a given model, resp. reverse engineering activities. Especially for a tool supported mapping of models there must exist a precise specification of the performed transformation [MM03]. As the MDA metamodel already indicates, it is essential to know about the metamodel of the source and target models to define mapping rules among the models.

The MDA doesn't specify or prescribe any language for the specification of these model transformations, but it recommends the UML, as a specification language for MDA models. Further the UML is the actual de facto standard for modeling in software development. Thus a transformation language for MDA models must be capable to transform object oriented models.

1.1.2 Integration of Heterogeneous Views on Development Models

Models in software engineering are usually not accessed directly but manipulated via views. A view is an abstraction of a model that focuses on a certain aspect like structure or behavior. Therefore a view usually provides a suitable description technique like static structure diagrams or activity graphs. There must be a specification that defines how graphical elements are mapped into a common model, like e.g. an instance of the UML metamodel. In UML this mapping is specified textual, the abstract syntax representation of the graphical notations are fragments of this instance model.

This approach lead to a rather unhandy UML metamodel with a structure that is motivated strongly by the constructs of its graphical notations. Another approach that is currently realized within the KOGITO project is the definition of a notation's semantics by the specification of a mapping of its abstract syntax representation into a common model. Figure 1.1 depicts this situation. This approach decouples the conceptual model from the used notations and thereby allows a much more clear conceptual model.

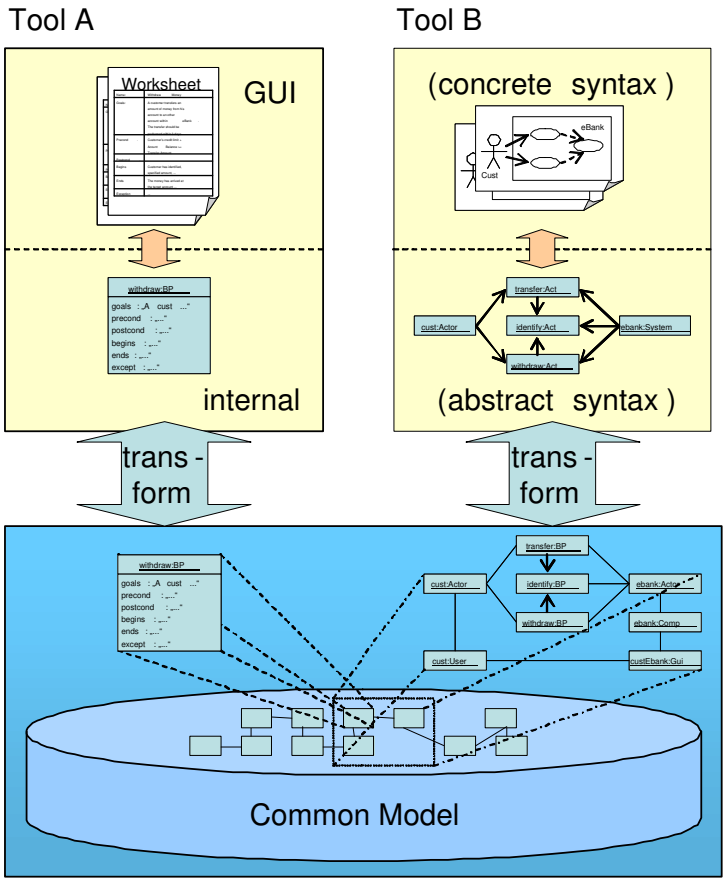

Figure 1.1: Integration of various views into a common model

1.1.3 Integration of Heterogeneous Applications

A classical area in software engineering is the integration of heterogeneous applications. Beyond the orchestration of different workflows the integration of heterogeneous application data models is an essential prerequisite for this task. The still increasing popularity of themes like B2B, B2C, and Web-Services reflect the relevance of this issue.

In this field XML [BP98] established as a textual standard exchange format. In contrast nearly all of the modern systems internally represent their data by object-oriented models, using languages like Java or C#. Therefore standards like XMI [XMI99] define a schematic mapping from class diagrams to XML document type definitions or XML Schemas [XML02]. This allows one to transform the internal representation of application data into a standardized XML representation which enables the use of general document transformation techniques like e.g. XSL. Thus XSL [XSL99] may be used for the integration of models.

Unfortunately those transformation techniques require to specify the transformation on a technical level. Ideally the specification of a transformation should be defined in terms of a more abstract application data model.

Other technologies like CORBA [OMG00] don't provide any transformation specification techniques. So the transformations have to be manually coded which is naturally error-prone, time, and cost expensive.

1.2 Related Work

XSL transformations [XSL99] are widespread as a language for the specification of transformations of XML documents into other textual representations. Thus XSL transformations are often used for the integration of applications that exchange different kinds of XML documents.

Unfortunately XSLT is not really intuitive and doesn't provide any graphical notation. XSLT specifications are quite verbose like most XML related technologies. Especially when using XSLT for transformations of the XMI representations of MDA models writing MDA transformations is a long winded and error-prone task. Therefore other approaches, as for example [PBG01], propose different XSL-based solutions, which help to develop more abstract model transformation specifications.

However, XSL does not provide any techniques to ensure or prove that newly generated models are actually conform to their metamodel. This must still be verified manually by the developer of the XSL document. Further XSL allows only the specification of unidirectional transformations.

Different techniques for the specification of model transformations are examined in [Ake00] and [GLR+02]. The most promising stem from the area of graph grammars [Roz97, Sch94, Nag96]. They deliver a theoretical foundation for the transformation of graphs. As a theoretically founded approach they have to be adopted to the concepts of object orientation. Therefore attributed,

typed graph grammars extended by constraints might be used for a deep integration with object orientation which missing up to now.

Other approaches to model transformation stem from specific domains like database migration and especially schema evolution [BB95, Cas95, LCC94, VW95]. These approaches show how to transform schemas systematically, so that models stored in an old version of a schema can be transformed automatically into a new schema. These often very technical approaches seem to be unhandy for specifying abstract model transformations especially, if also an inverse transformation is needed.

1.3 BOTL

Hence, we are convinced that neither algorithmic languages nor graph grammars are appropriate and ergonomic languages, to define object model transformations. Therefore we propose BOTL, the Bidirectional Object oriented Transformation Language, as a trade-off between these two approaches.

In BOTL we regard structures that consist only of typed objects and associations. While structural transformations are expressed by rules that are similar to those in graph grammars we use algorithmic expressions within these rules to relate the values of attributes and object identities.

This paper introduces the BOTL that is currently developed by the authors. BOTL comes with a sound, mathematical foundation of the language itself and its transformation mechanisms. BOTL offers the ability to use graphical description techniques and algorithmic descriptions integrated to graphically define a set of mapping rules. These rules determine the relation between two metamodels and how the according models are transformed. Further BOTL allows reasoning about the following properties:

Applicability, i.e. the property that the application of rule set for a given metamodel doesn't cause any conflicts for any arbitrary source model;

Metamodel Conformance, i.e. the property that the application of a rule set can only generate target models that are conform to a given target metamodel;

Bijectiveness of transformations, i.e. the possibility to invert the rules of a rule set to retranslate models (respectively a part of a model), so that the source model is obtained again or the source model and the re-transformed model are isomorph.

This technical report is structured as follows. In Section 2 a small scenario is introduced that serves as a running example throughout this report.

Section 3 introduces the BOTL formalism and its underlying concepts. A mathematical model for class models, object models, and rule sets is described together with a UML-based notation. Further the mechanism of a rule set application is formally defined.

Section 4 presents two basic properties of BOTL rule sets: applicability and metamodel conformance. Further the concept of bijectivity of rule sets is introduced. A set of techniques to prove these properties is provided that make only use of a given rule set together with its source and target metamodel. These verification techniques allow a tool supported proof of these properties of rule sets.

Section 5 introduces two extensions of the basic BOTL formalism. First an extension for the support of inheritance is defined, which allows one to use BOTL together with metamodels that contain inheritance relations. Further a mapping of BOTL metamodels to UML class models is defined. Thus we show that BOTL is not restricted to models that are defined in terms of the BOTL formalism but it can be also used to directly transform UML models.

The report is rounded up by a short discussion and outlook on future work in Section 6. See also [BM02] for a short BOTL introduction.

2 Running Example

In this section a small example is introduced to illustrate the proposed approach and guide the reader through the paper. The example is intentionally small but it covers many relevant aspects.

In the sample scenario there are two applications, the Alpha Information System and the Beta Application, that both process data about employees and offices. The structures of the object models that the two applications use internally are determined by UML class diagrams.

We use the notion metamodel within the example despite the fact that those class diagrams are usually named models within the MOF hierarchies. Note that the confusion stems from the fact that metamodels within the MOF level 2 are depicted as class diagrams similar to models of the MOF level 1. So visually there is no difference. But metamodels of MOF level 2 usually describe modeling languages but not data models. Within BOTL we don't care about this unobvious difference between the meanings. So informally spoken every class diagram can be a metamodel.

The class diagram for the first application, the Alpha Information System (AIS), is depicted in Figure 2.1. Some of the class attributes are written in bold face to indicate that their values serve as primary keys to identify objects. We will discuss this issue later on in detail, so far it isn't of any relevance, yet.

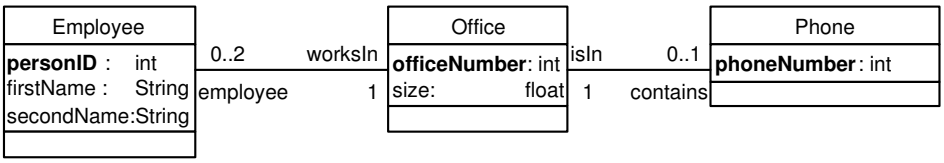

Figure 2.1: The metamodel of the Alpha Information System

As one can see every employee works in exactly one office, while an office offers space for up to two employees. Further every office may have a phone.

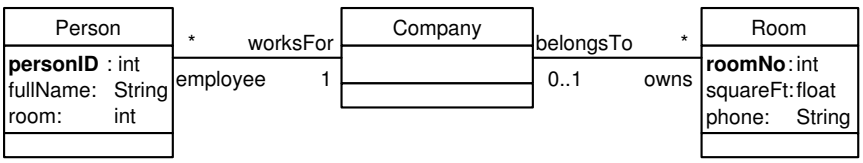

Figure 2.2: The metamodel of the Beta Application

Figure 2.2 shows the Beta Application's class diagram for dealing with employees and offices. Obviously the Beta Application has no extra class for phones; instead the phone number is stored in an additional attribute of the class Room.

It's easy to recognize, that the two metamodels try to express the same or at least very similar concepts in a common application domain. But to allow inter operation between the two applications all exchanged data either has to be converted in a format according to one of the partners' metamodel format or a common metamodel might be chosen. A common metamodel makes sense if you want to integrate a vast number of applications, but in this example it's assumed that the AIS is to be extended to be capable of communicating with the BA.

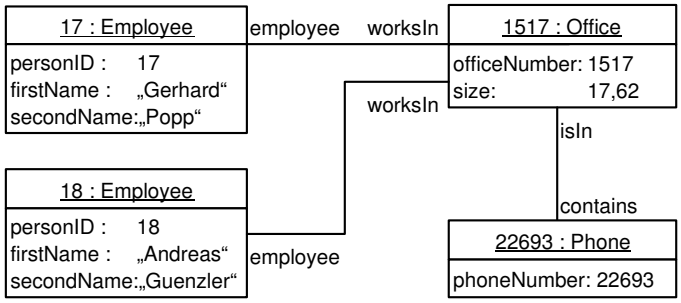

Figure 2.3: The alpha object model m_α that is conform to the AIS metamodel

Figure 2.3 shows a sample object model of the Alpha Information System that should be transformed to correspond to the Beta Application's meta model. We will refer to this model as the alpha model m_α.

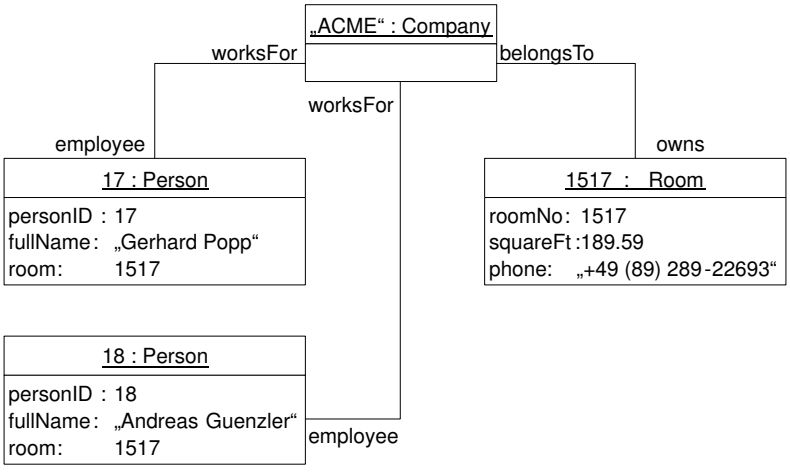

Figure 2.4: The transformed beta object model m_β that originates from the AIS model m_α

Intuitively one can sketch a solution by interpreting the names of associations, classes, and attributes. The result of a transformation of the the model from Figure 2.3 into an instance m_β of the metamodel shown in Figure 2.2 should obviously look like depicted in Figure 2.4. As one

can see the resulting model had to be enriched with static information, like e.g. the the local area code, that is not available in the AIS but required within the Beta Application.

The goal of the proposed BOTL approach is to provide a language that allows a developer to easily express this intuitive knowledge about how the two models relate. Further the language needs well defined semantics that allows the generation of model transformers.

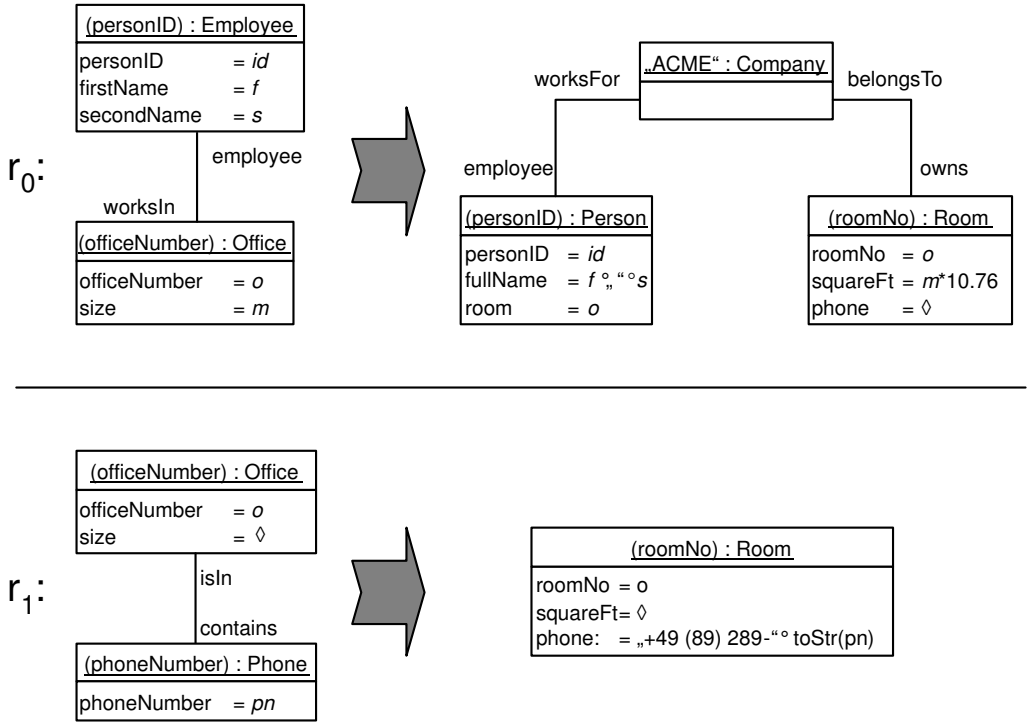

Figure 2.5: Sample BOTL rule set $r = (r_0, r_1)$

Therefore a user may write BOTL rules that define model transformations. For the transformation of Alpha into Beta models one might write two BOTL rules r_0 and r_1 as depicted in Figure 2.5. The left side of a rule consists of a model pattern that is searched in the source model. In our example the source model is the Alpha model of Figure 2.4. The right side consists of a model pattern that is created within the target model. The attribute values and identifiers of newly created objects can be calculated from those found in the source model matching. Therefore the two model patterns contain constants, terms, and variables instead of attribute values and identifiers.

Informally spoken the first rule r_0 identifies employees and their offices and relates them to a pair of Person and Room objects in the Beta model. The Company object “ACME” of the Beta model remains always constant.

Note that the identity of objects is determined by relating them with an attribute value. This allows to access already generated objects in the target model. If the identity of an object was not determined in a rule a unique identifier would be generated instead. Further there are also

more enhanced strategies for dealing with object identities, such as to determine the identity by a mathematical term, that imply a much higher complexity and thus lie out of the scope of this paper.

Below r_0 in Figure 2.5 the second BOTL rule r_1 is shown. This rule identifies objects of the type Office and their belonging Phone objects in the alpha model and creates a corresponding Room object in the beta model.

If r_0 has already been applied the Room object may already exist with an $\diamondsuit$ value for the attribute phone. In this case it will just be updated. The author of the rules can add static information to the rules. So he can provide the full phone number for the Beta Application, while the AIS does not have any information about the country and area code of the phone numbers.

Now we explain the application of the given sample rule set in more detail. The first step in the application of a single rule is to identify a match of its left side in the source model.

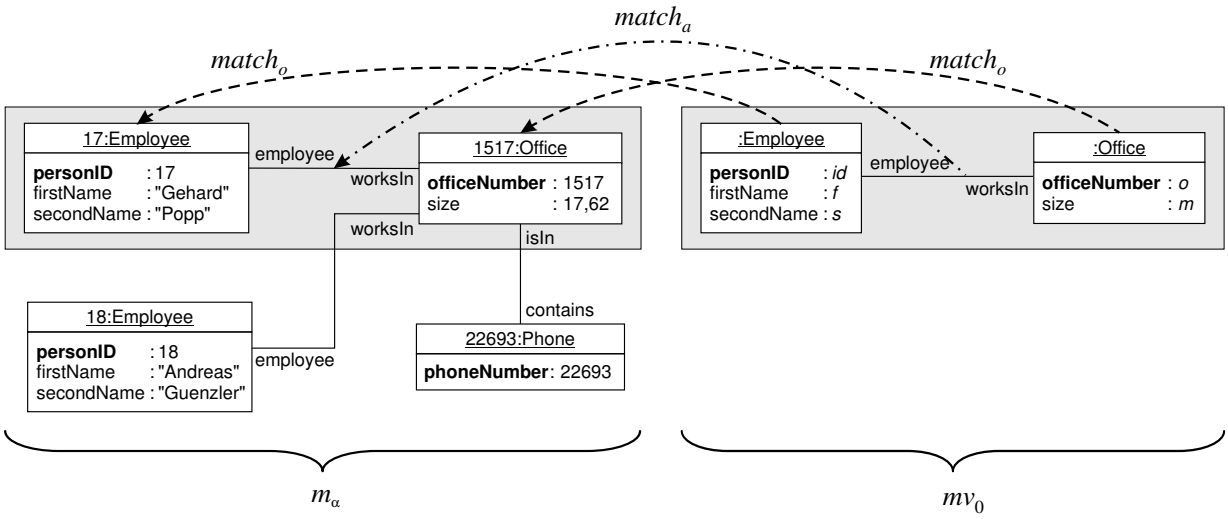

Figure 2.6: r_0's left side model pattern matches in the model m_α

Figure 2.6 shows how the left hand model pattern of the rule r_0 matches to a model fragment of the model m_α (c.f. Figure 2.3). A valid match implies that we find the structure of our model pattern as a substructure of the source model. I.e. the structure must be congruent and we can map every element of the model pattern to an object or association of our source model, as indicated in the figure.

For every match in the source model a new model fragment is created that will be merged into the target model m_β. Figure 2.7 shows how the model fragment that was found in Figure 2.6 is transformed into a new one conforming to the metamodel of the Beta Application. As one can see the attribute values and identifiers are calculated in accordance to the terms, variables and constants of the model patterns in the rule r_0. The resulting fragment is depicted on the right side of the figure.

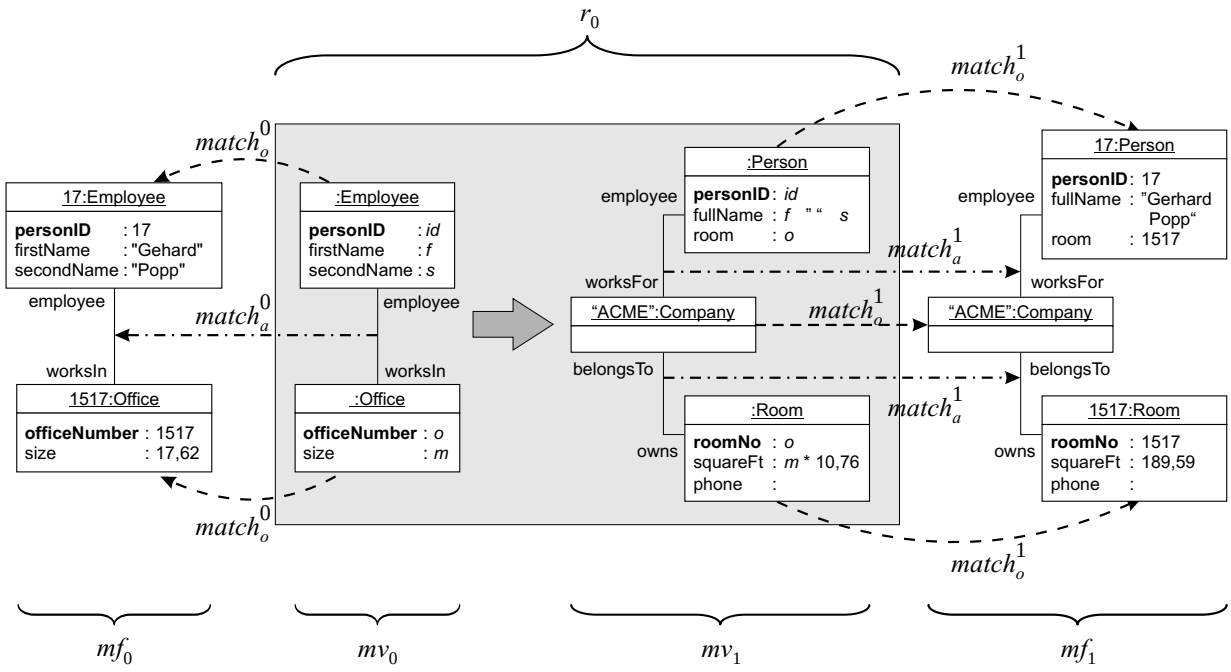

Figure 2.7: A new model fragment of the Beta model is created from a match of r_0's left hand side.

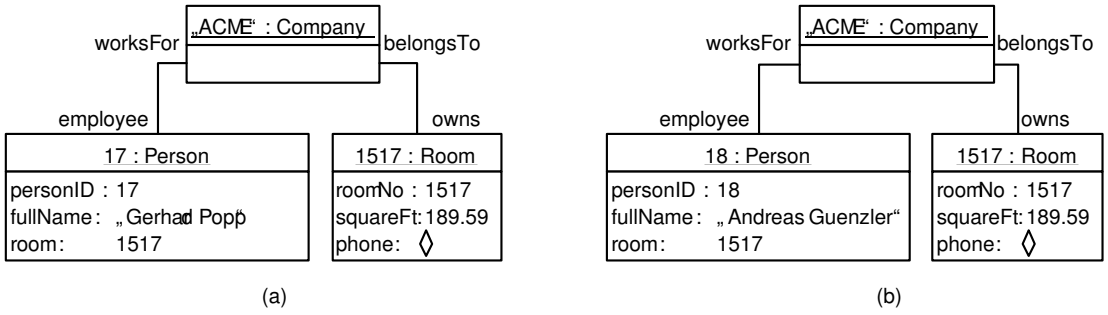

Figure 2.8: The model fragments created by the application of r_0

Obviously there are two possible matches of r_0's left side in the model m_β. Applying r_0 for the second match produces another model fragment for the target model. Both model fragments are shown in Figure 2.8. In both fragments the attribute value for phone contains an $\diamondsuit$. This denotes that this attribute value is not yet determined.

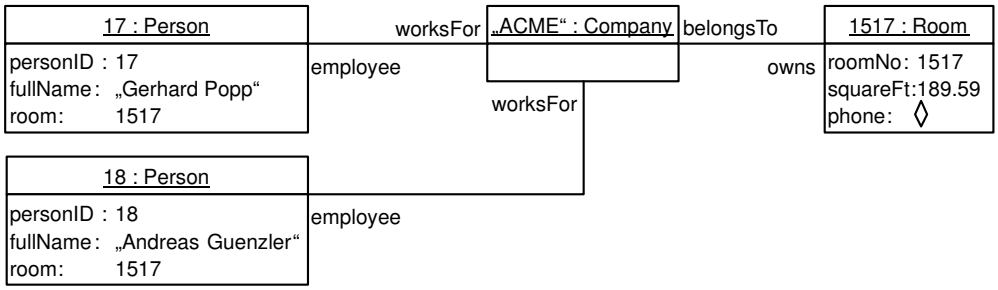

Figure 2.9: The result of merging the two model fragments that have been created by the application of r_0.

All the created model fragments are merged to one big fragment that finally should look like our target model m_β in Figure 2.4. Figure 2.9 shows the new model fragment that results from merging the two newly created model fragments from Figure 2.8. Since there are only two possible matches in the source model, this fragment is already the result of the application of r_0. Obviously it still does contain a $\diamondsuit$ in the phone attribute of the Room object. This value might be overwritten during a subsequent rule application. If it still does remain after the application of the entire rule set, it will be replaced afterward by a default value like e.g. an empty string.

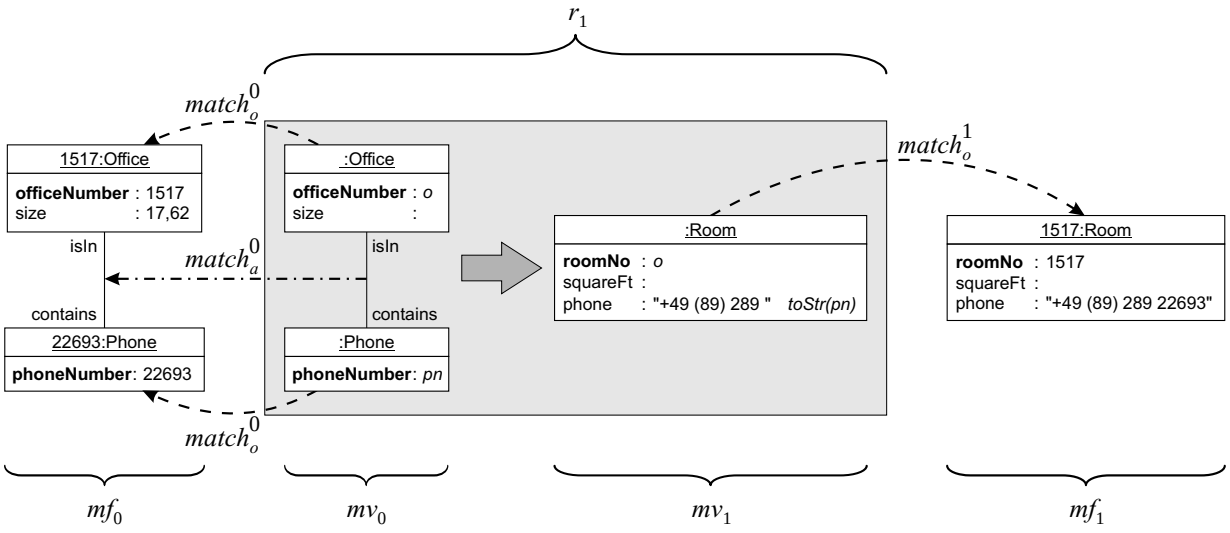

Figure 2.10: The second rule r_1 generates a model fragment that consists of only one object.

The application of rule r_1 yields to one match of the left hand rule side in the model m_α. Figure 2.10 shows how the data of the matching model fragment is used to create a new model fragment that consists only of one object of the type Room.

The bold typeface that is used for the roomNo attribute of the Room object in the rule indicates that this attribute servers as a primary key to uniquely identify objects of this type. As one can see the new Room object has the same value for the attribute roomNo as the one created by the rule r_0. Thus we know that the two Room objects created by r_0 and r_1 represent actually the same object in the target model. Consequently they are merged into one object whereby the existing $\diamondsuit$ value of the phone attribute is overwritten by the value "+49 (89) 289 22693" of the object generated by r_1. The resulting model fragment is the target model m_β depicted in Figure 2.4.

The mechanism of the rule based model transformation that was described here in a very informal fashion is defined formally in the following section. Further it is interesting to know for a given rule set, if the application of the rules might fail, e.g. because we try to overwrite the squareFt value generated in our sample rule r_0 with a different value resulting from another rule. Thus Section 4.1 does show how one can assure that a given rule set is applicable. Another property we didn't discuss yet is the ability of a rule set to create only metamodel conform models, i.e. models that are conform to the given target metamodel. In this example the generated model m_β is conform to the rules target metamodel as one can easily verify. Section 4.2 provides verification techniques to formally prove that this is the case for arbitrary source models.

3 BOTL Formalism

The Bidirectional Object-oriented Transformation Language (BOTL) is formally defined based on simple set theory. Currently BOTL serves mainly as a specification mechanism for the description of tool chains and the integration of development models. E.g. within the FORSOFT II project Automotive [vBBRS02] BOTL is used to define the transformation of data between the CASE tools DOORS, The UML Suite, and ASCET-SD. This specification is then implemented manually as the interfaces to the CASE tools are proprietary. in the Project KOGITO the BOTL is used to unambiguously define the mapping of various description techniques into a common conceptual development model. As the UML and especially class diagrams are widely used, BOTL is based upon a formalization of UML class diagrams. This formalization is given in Section 3.1. Section 3.2 describes the formalism used for rules, rule sets, and their application. Section 3.3 introduces a UML-based representation for rules and meta models. In Section 3.4 the semantics of rule set applications are formally determined.

3.1 Basic formalism

In this section a formal model for class models and object models is presented. It is our goal to capture all aspects of UML class and object diagrams that concern model transformations in the given formalization . However one main concept of object orientation, namely inheritance, is not yet introduced within this section since it is discussed in section 5.2.

First we define the set of all possible identifiers. Identifiers are needed to distinguish classes, objects and primitive types.

Definition 3.1.1 (Set of identifiers $\mathbb{ID}$)
Let $\mathbb{ID}$ be the set of valid *identifiers* and $|\mathbb{ID}| = \infty$. ❍

A primitive type can be referenced by its identifier and contains a (possibly infinite) set of values.

Definition 3.1.2 (Type)
A *type* is a tuple (it, T_{val}) consisting of

- a *type identifier* $it \in \mathbb{ID}$ and
- a set T_{val} of *type elements*.

❍

In programming languages like Java typical primitive types are int (not Integer), float, etc. Regarding our transformation formalism other types like the classes String or Integer may be regarded as primitive, too.

Definition 3.1.3 (Type Allocation (*TA*))
A *Type Allocation TA* is a set of types $\{t_0, \ldots, t_n\}$ with:

$$\forall t_i, t_j \in TA: \quad t_i|_{it} = t_j|_{it} \Rightarrow t_i = t_j \tag{3.1}$$

❍

Thereby (3.1) ensures that all types have unique identifiers.

Thus a type allocation is a set of types with different identifiers. However there might exist two types in a type allocation that have common elements. For example the value 42 might be an example for an primitive value that is contained in the set values of the type int and that of the type long. In Java there would exist two values 42 and 42L that must be converted to the desired type.

The following definition formalizes classes according to the UML notation. Concepts that are not relevant for our mechanism, like e.g. methods or abstract classes, are left out. Additionally a set of attributes can be marked as primary keys, which means that these attribute values determine the identity of the class instances.

Definition 3.1.4 (Class (*c*))
A *Class c* is a tuple $(id, A, Keys)$ consisting of

- an identifier $id \in \mathbb{ID}$,
- a set A of attributes (n,t) consisting of
  - an identifier $n \in \mathbb{ID}$ and
  - a type $t \in TA$

  with the property:

  $$\forall (n_i, t_i), (n_j, t_j) \in A: \quad n_i = n_j \Rightarrow t_i = t_j \tag{3.2}$$

- a set *Keys* of tuples (n,t) with $Keys \subseteq A$

Let *consistent* be a predicate defining if a class is consistent to a type allocation[1].

$consistent(c, TA) :\Leftrightarrow \forall t \in c|_A|_t : t \in TA$

❍

Thereby (3.2) ensures that every attribute of a class has a unique name.

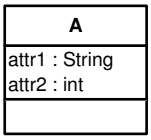

Figure 3.1: A sample class

In Figure 3.1 a class A with two attributes attr1 and attr2 can be seen. A restricted form of UML class diagrams is used for the representation of metamodels that will be presented in detail in Section 3.3. The tuple $(A, \{(attr1, String), (attr2, int)\}$ represents this class.

Definition 3.1.5 (Class Allocation (*CB*))
A *Class Allocation CB* consists of a set of classes $\{c_0, \ldots c_n\}$ for that holds:

$$\forall c_i, c_j \in CB : \quad c_i|_{id} = c_j|_{id} \Rightarrow c_i = c_j \tag{3.3}$$

The predicate *consistent* holds if a class allocation is consistent to a given type allocation.

$consistent(CB, TA) :\Leftrightarrow \forall c \in CB : consistent(c, TA)$

❍

According to (3.3) the identifiers of the classes have to be pairwise different.

Different classes may be related by associations. Within BOTL only some aspects of associations between classes are relevant. These are the associated classes, their role names, and the multiplicities of the association. Only binary associations are considered.

Definition 3.1.6 (Class Association (*AE*))
A *Class Association AE* regarding a class allociation *CB* is a set of tuples $ae = (rn, c, m, t, nav)$, so called *class association ends*, consisting of

- a *role name* $rn \in \mathbb{ID}$,
- a class $c \in CB$,

[1]For a set T of tuples $t = (x,y)$ let $T|_x$ the set $\{t|_x : t \in T\}$.

- a *multiplicity* $m \in (\mathbb{N}_0 \cup \infty) \times (\mathbb{N} \cup \infty)$,
- an *aggregation type* $t \in \{\text{none}, \text{aggregate}, \text{composite}\}$, and
- a boolean navigability property *nav*,

so that it holds:

$$1 \leq |AE| \leq 2 \tag{3.4}$$

$$\wedge\ (|AE| = 2 \wedge AE = \{(rn_0, c_0, m_0, t_0, nav_0), (rn_1, c_1, m_1, t_1, nav_1)\} \wedge \exists_1 i \in \{0,1\} : t_i = \text{none})$$

$$\vee(|AE| = 1 \wedge AE = \{(rn, c, m, t, nav)\} : t = \text{none}) \tag{3.5}$$

The predicate *consistent* holds if a class association is consistent to a given class allocation.

$$consistent(AE, CB) :\Leftrightarrow \forall c \in AE|_c : c \in CB$$

❍

(3.5) ensures that at least one of the ends is of the aggregation type none to prevent mutual compositions or aggregations of an association. Thereby $|AE| = 1$ denotes an association as it can be seen in Figure 3.4. The navigability property *nav* has a purely informal character within the formalism.

An association of the type aggregate defines an asymmetric, acyclic relation between objects. So an object may not directly or indirectly aggregated by itself. An association of the type composite is even stronger. An object can have at most one container in any composite relation. Within BOTL both aggregation types have only informal character. Aggregation type is only included for later extensions.

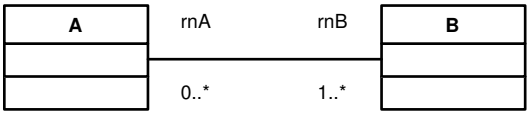

Figure 3.2: A class association

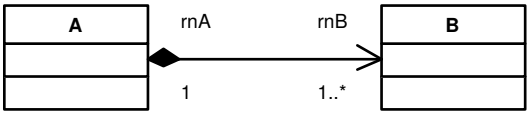

Figure 3.3: An unidirectional navigable composition

In Figure 3.2 a class association between the two classes A and B can be seen. This class association is denoted by the set $AE_1 = \{(rnA, A, \text{none}, (0, \infty), true), (rnB, B, \text{none}, (1, \infty), true)\}$. Figure

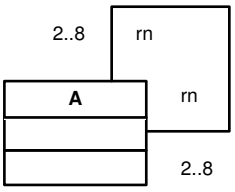

Figure 3.4: A symmetric class association

3.3 shows a unidirectional (in direction of the arrow) navigable composition, consisting of the set of class associations ends $AE_2 = \{(rnA, A, \text{composite}, (1,1), false), (rnB, B, \text{none}, (1,\infty), true)\}$. A symmetric class association, as it is needed e.g. for modelling an association of the type "Person is_related_to Person", is depicted in Figure 3.4. The set $AE_3 = \{(rn, A, \text{none}, (2,8), true)\}$ represents this association.

Definition 3.1.7 (*oppositeEnd*(AE, ae))
The function *oppositeEnd* returns the appropriate opposite end of *ae* within a class association *AE*.

$$oppositeEnd(AE, ae) = \begin{cases} ae & \text{for } |AE| = 1 \\ ae^* & \text{for } |AE| = 2 \text{ with } AE = \{ae, ae^*\} \\ \perp & \text{otherwise} \end{cases}$$

❍

Within the BOTL formalism and its application the notion metamodel is used more generally than it is used by the UML or MOF community. In BOTL a metamodel is used for the definition of a data model that may be used within a tool. This may be e.g. the UML metamodel used within a UML CASE tool. But from the BOTL point of view also a class diagram as e.g. shown in Figure 2.1 is called a metamodel. In the UML community this is called a model, as it is simply an instance of the UML metamodel. This confusion stems from the fact, that class diagrams are used at different MOF levels.

Definition 3.1.8 (Metamodel (*mm*))
A *metamodel mm* is a tuple (TA, CB, CA) that consists of

- a type allocation *TA*,
- a class allocation *CB* with $consistent(CB, TA)$, and
- a set of class associations *CA* with:

  $\forall AE \in CA : \forall ae \in AE : ae|_c \in CB$

  I.e. the ends of all associations reference classes c_i of the class allocation *CB* of the metamodel.

❍

Instances of classes are called objects. Within BOTL a metamodel has to be defined before objects may be instantiated. Every object has an unique identity, which is one key issue within object orientation. If the according class defines primary keys, then the values of the corresponding attributes of the object determine together the unique identity.

Definition 3.1.9 (Object (o))
An *object* o is a tuple (oi, ot, V) consisting of

- an identifier $oi \in \mathbb{ID}$,
- an *object type* $ot \in mm|_{CB}$, and
- a set V of tuples (a, v) with:

$$ot|_A|_n = V|_a \ \wedge \ \left|ot|_A\right| = \left|V\right| \tag{3.6}$$

$$\tag{3.7}$$

The predicate *consistent* holds if an object is consistent to a given metamodel.

$$consistent(o, mm) :\Leftrightarrow o|_{ot} \in mm|_{CB} \tag{3.8}$$

$$\wedge \forall (a, v) \in o|_V : \exists t : \quad (a, t) \in ot|_A \tag{3.9}$$

$$\wedge \ v \in t|_{T_{Val}} \cup \{\diamondsuit\}$$

For objects that are instances of classes with primary keys it further has to hold:

$$o|_{ot}|_{Keys} \neq \emptyset \Rightarrow it\text{exists } one \text{ bijective mapping } pK \text{ for all objects of the } typeo|_{ot} \tag{3.10}$$

For a convenient access to attribute values we introduce the following notation:

$$o.att := \begin{cases} v & \text{if } \exists (a, v) \in o|_V : a = att \\ \perp & \text{else} \end{cases}$$

An object o with: $\forall att \in o|_V|_a : o.att \neq \diamondsuit$ is called $\diamondsuit$-*free*. ❍

(3.6) uses the projective $|$ operator on sets to ensure that the object has exactly the attributes specified in the class definition. (3.9) determines that the attribute values of an object are exactly of the type defined in the according class. (3.10) enforces, that the identity of an object with primary key attributes is linked to these values by stating that there must be a bijective mapping that allows to calculate the object identifier from these attributes. Since in practice this function is not always of interest we can assume that it yields to a tuple of the key attribute values as an unambiguous object identifier.

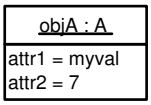

Figure 3.5: A sample object

In Figure 3.5 an object objA with two attributes attr1, attr2 and their values can be seen. The object type of this object, i.e. its according class, is depicted in Figure 3.1. Within the BOTL formalism the object is represented by the tuple $(objA, A, \{(attr1, "myval"), (attr2, 7)\})$. Thereby objA.attr1 results in "myval".

Definition 3.1.10 (Object Allocation (*OB*))
A *object allocation OB* is a set of objects $\{o_0, \ldots, o_n\}$ for that holds:

$$\forall o_i, o_j \in OB: \quad o_i|_{oi} = o_j|_{oi} \Rightarrow o_i = o_j \tag{3.11}$$

The predicate *consistent* holds if an object allocation is consistent to a given metamodel.

$$consistent(OB, mm) :\Leftrightarrow \forall o \in OB : consistent(o, mm)$$

An object allocation *OB* is called $\diamondsuit$*-free,* if all objects *o* of the object allocation are $\diamondsuit$-free. ❍

All objects of an object allocation have an unique identifier (3.11).

Similar to classes objects may be connected by object associations. As objects are instances of classes, object associations are instances of class associations.

Definition 3.1.11 (Object Association (*oa*))
An *object association oa* regarding an object allocation *OB* is a tuple $(OAT, card, OAE)$ consisting of

- a class association $OAT \in mm|_{CA}$,
- a *cardinality* $card \in \mathbb{N}$, and
- a set *OAE* of tuples $oae = (ae, o)$ that consist of
  - a *class association end ae* and
  - an object $o \in OB$

for that holds:

$$1 \leq |OAE| \leq 2 \wedge OAE|_{ae} = OAT \quad \wedge \tag{3.12}$$
$$\forall oae \in OAE : oae|_{ae}|_c = oae|_o|_{ot} \tag{3.13}$$

The predicate *consistent* holds if an object association is consistent to an object allocation.

$$consistent(oa, OB) :\Leftrightarrow \forall o \in oa|_{OAE}|_o : o \in OB$$

❍

Within an object association the ends in the set *OAE* and the according class association *OAT* coincide (3.12). (3.13) implies that the objects *o* of the object association are of the correct type.

This definition of associations does not yet guarantee that object associations have to be consistent with the metamodel *mm*, since eventually cardinalities may to exceed the valid limits.

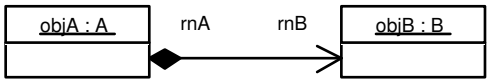

Figure 3.6: A sample object association

In Figure 3.6 an object association that matches to the class association in Figure 3.3 is depicted. It is represented by the tuple

$$(AE_2, 1, \{((rnA, A, (1,1), comp), objA), ((rnB, B, (1,\infty), nav), objB)\})$$

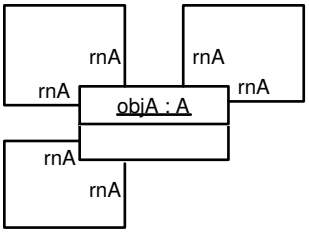

Figure 3.7: A symmetric object association

$|OAE| = 1$ represents an object association as it can be seen in Figure 3.7. This object association is conform to the class association from Figure 3.4. The object association is represented by the tuple $(AE_3, 3, \{((rnA, A, (2,8), std, objA))\})$. As one can see the three association instances in the UML diagram are treated as one single association with a cardinality of 3. An object association has no identifier and consequently no own identity. Thus it is not possible to distinguish two object associations of the same type between the same two objects. To represent multiple object associations of the same type among two objects the cardinality *card* of an object association is used in the BOTL formalism.

A model consists of instances of the according types defined in a metamodel. So a model refers to a metamodel and contains sets of objects and object associations.

Definition 3.1.12 (Model (m))
A *model m* is a tuple (mm, OB, OA) consisting of

- a metamodel *mm*,
- a $\Diamond$-free object allocation *OB* with $consistent(OB, mm)$, and
- a set of object associations *OA* with $\forall oa \in OA : consistent(oa, OB)$

for that holds:

$$\forall AE \in mm|_{CA} : \forall ae \in AE \text{ with } (lb, ub) := oppositeEnd(AE, ae)|_m :$$
$$\forall o \in OB \text{ with } o|_{ot} = ae|_c$$
$$lb \leq \sum_{oa \in OA \text{ with } oa|_{oat} = AE \wedge (ae,o) \in oa|_{OAE}} oa|_{card} \leq ub \tag{3.14}$$
$$\forall oa \in OA : \forall oae \in oa|_{OAE} :$$
$$oae|_o \in OB \wedge oae|_{ot} \in mm|_{CA} \tag{3.15}$$

❍

(3.14) determines that there is a correct number of object associations *OA* for all class associations *CA* of the metamodel *mm*, if according objects are contained in the model *m*. Therefore in (3.14) the sum of all outgoing object associations is build. E.g. a metamodel may demand that all objects of type A must have at least two attached associations of a certain kind but must not have more than eight of them (as depicted in Figure 3.4). If this consistency criteria does not hold for a set of objects and associations the statement (3.14) will be violated.

(3.15) determines that all object associations *OA* connect only objects contained in the model *m*. That means also that there are no dangling associations. The property that associations connect only objects of the correct type is not enforced here, because this is already ensured by the definition of object associations in (3.13).

Definition 3.1.13 (Set of Conform Models ($\mathbb{M}_{mm}$))
$\mathbb{M}_{mm}$ denotes the set of all valid models that are conform to the metamodel *mm*. ❍

3.2 BOTL Transformation Rule Sets

BOTL uses rules for the definition of a model transformations. Each rule defines a mapping of a clipping of the source model onto a clipping of the target model. Each rule consists of a left and a right hand side model variable. In the following the constituents of rules are defined.

Within model variables terms are used for the description of the interrelationship of source and target model attribute values and identifiers. BOTL does not contain a complete definition of correct terms and their semantics. Within the pragmatic context of BOTL this is not needed, as "correct" terms used in BOTL are defined with some programming language in mind.

Definition 3.2.1 ($Term_{TA}$)
$Term_{TA}$ denotes the set of all valid terms over types of the type allocation TA. $Term_{TA}$ contains $\diamondsuit$. The set of possible types of a term is determined by: $type : Term_{TA} \rightarrow \mathscr{P}(TA)$ ❍

Definition 3.2.2 ($Term_{\mathbb{ID}}$)
$Term_{\mathbb{ID}}$ denotes the set of all valid terms over the set of identifiers $\mathbb{ID}$. Thus $Term_{\mathbb{ID}}$ may consists of

- a constant (element of $\mathbb{ID}$),
- a variable,
- a tuple of variables, or
- $\diamondsuit$

In the case that the identifier consists of a tuple of variables there has to exist a bijective mapping uK that maps the values of the variables to $\mathbb{ID}$. Thereby uK generates identifiers different from all constants, variables, and $\diamondsuit$. ❍

Examples for valid identifier terms are: $\diamondsuit$, id, (cid, aid)

Object variables describe objects. They are depicted similar to objects (s. Fig. 3.8). They contain terms for the identity and the attribute values of objects. For better readability attributes that serve as primary keys can either be listed in the object identifier (see Section 3.3) or printed bold in the object variables attributes box as depicted in Figure 3.8.

Definition 3.2.3 (Object Variable *ov*)
An *object variable ov* is a tuple $(ovid, oiv, otv, VV)$ consisting of

- an unique object variable identifier *ovid*,
- an object identifier $oiv = \begin{cases} term \in Term_{\mathbb{ID}} & \text{if } ov|_{otv}|_{Keys} = \emptyset \\ \varepsilon & \text{otherwise} \end{cases}$
- an object type *otv*, and

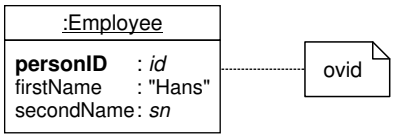

Figure 3.8: A sample object variable

- a set VV of tuples (a,t) with:

$$\begin{aligned}\forall(a,t)\in VV: \quad & t\in Term_{TA}, a\in\mathbb{ID}\\ & \wedge\, otv|_A|_n = VV|_a\\ & \wedge\, \left|otv|_A\right| = \left|VV\right|\\ & \wedge\forall(a,t)\in VV: \quad \exists(a,p)\in otv|_A \wedge p\in type(t)\end{aligned} \tag{3.16}$$

The predicate *consistent* holds if an object variable is consistent to a given metamodel.

$consistent(ov,mm) :\Leftrightarrow ov|_{otv}\in mm|_{CB}$

For a convenient access to attribute term we introduce the following notation (similar to the notation of Def. 3.1.9):

$$ov.att := \begin{cases} t & \text{if } \exists a:(a,t)\in ov|_{VV}\wedge a=att\\ \perp & \text{else}\end{cases}$$

❍

Please note that the object variable identifier is usually hidden but formally necessary to allow to distinguish several object variables with the same content (except their *ovid*s) within an object variable allocation as defined in Definition 3.2.4. Further a term as an object identifier can also be $\diamondsuit$. In this case the identity of the object variable is not determined by a specific value. (3.16) determines that terms have to be specified for every attribute of an object variable and that each term's type has to match to the type of the corresponding attribute.

In Figure 3.8 a sample object variable belonging to the metamodel of Figure 2.1 is shown. In the example the used terms are variables depicted in italic face, or constants. In the example no term for the identifier is used as Employee has primary key attributes which are shown in bold face. To refer to the object variable its name can be shown optionally as a note as shown in the figure.

Definition 3.2.4 (Object Variable Allocation *OVB*)
An *object variable allocation OVB* is a set of object variables *ov* for that it holds:

$\forall ov_i, ov_j\in OVB: (ov_i|_{ovid} = ov_i|_{ovid} \Rightarrow ov_i = ov_j)$

The predicate *consistent* holds if an object variable allocation is consistent to a given metamodel.

$consistent(OVB, mm) :\Leftrightarrow \forall ov \in OVB : consistent(ov, mm)$

❍

Object variable associations describe object associations. Graphically they are depicted like object associations.

Definition 3.2.5 (Object Variable Association *ova*)
An *object variable association ova* is a tuple $(OVAT, cardv, OVAE)$ consisting of

- a class association $OVAT$
- a cardinality $cardv \in \mathbb{N}$
- a set $OVAE$ of tuples $ovae = (ae, ov)$ that consist of
  - a class association end ae
  - an object variable ov

  for that it holds:

$$\begin{aligned} &\{ae : (ae, ov) \in OVAE\} = OVAT \\ \wedge\ &\forall ovae \in OVAE : ovae|_{ae}|_c = ovae|_{ov}|_{otv} \\ \wedge\ &1 \leq |OVAE| \leq 2 \end{aligned}$$

The predicate *consistent* holds if an object variable association is consistent to a given object variable allocation.

$consistent(ova, OVB) :\Leftrightarrow ova|_{ov} \in OVB$

❍

Model variables are the left hand sides and the right hand sides of rules. Model variables consist of object variables connected by object variable associations.

Definition 3.2.6 (Model Variable *mv*)
A *model variable mv* is a tuple (mm, OVB, OVA) that consists of

- a metamodel mm
- an object variable allocation OVB with $consistent(OVB, mm)$, and
- a set of object variable associations OVA with
  $\forall ova \in OVA : consistent(ova, OVA)$.

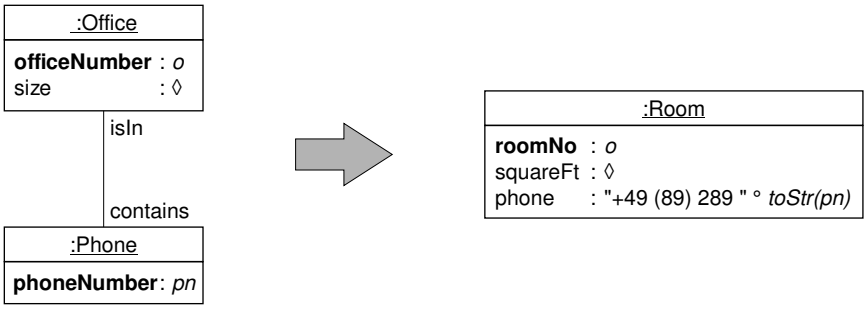

Figure 3.9: A sample rule

All object variables of a model variable must be connected to each other directly or indirectly by object variable associations. ❍

Thus, a model variable can be inconsistent regarding the metamodel *mm* because the cardinalities of the object variable association don't have to be conform to the multiplicities that are defined within *mm*.

Definition 3.2.7 (Model Transformation Rule r_i)
A *model transformation rule* r_i is a tuple (mv_0, mv_1) consisting of

- a source model variable mv_0 and
- a target model variable mv_1.

Thereby it has to hold: Every variable that occurs within terms of a rule must must occur at both sides of the rule. The LHS may not contain any constants as terms. $\diamondsuit$ is used as a free variable that occurs exactly once. Terms consisting of a tuple of variables for object identifiers that occur within a source model variable are treated like $\diamondsuit$ terms. All variables of these tuples have to be bound outside of tuples within the source model variable[2]. ❍

An example for a rule between the metamodels shown in Figure 2.1 and 2.2 can be seen in Figure 3.9.

For a better tangibility the model variable mv_0 is also called the *LHS* of the rule while mv_1 is called the *RHS* of the rule.

A $\diamondsuit$ value occurring on the RHS yields to target model fragments that contain an arbitrary value again represented by $\diamondsuit$. When merging such attributes with the value $\diamondsuit$ this value is always overwritten by any other value. If a variable occurs multiple times on the RHS but not on the LHS this would imply a constraint. The newly created target model fragment contains attributes whose values are not finally determined but the their values depend on each other.

[2]This predefinition is necessary to make the bidirectional application of rules possible, as this is defined later.

For example a fragment with an attribute i might be created where i might have an arbitrary value. Further it might have another attribute j that must always have the value $2 \cdot i$. By merging such fragments additional constraints like these might be added (e.g. it might hold additionally for the attribute j that $j = 3 \cdot k$). This results in an equational system that can be resolved only after the application of all rules. Hereby there is the danger that the equational system is not solvable and the model transformation is not applicable. Yet to ensure successful model transformations the constraint in Definition 3.2.7 prohibits such situations.

Definition 3.2.8 (Model Transformation Rule Set r)
A *model transformation rule set r* is a finite sequence of model transformation rules $r_0, ..., r_n$ for that it holds:

$$\forall k \in \{0,1\} : \forall i, j \in \{0, ..., n\} : r_i|_{mv_k}|_{mm} = r_j|_{mv_k}|_{mm} \tag{3.17}$$

(3.17) determines that all model transformation rules of a model transformation rule set define transformations between the same two metamodels. ❍

The following section concludes the UML based graphical specification language that is build on top of BOTL.

3.3 UML-based Notation for BOTL

In the previous section we already introduced exemplary a UML-based graphical notation for BOTL metamodels and model variables. This section defines a small UML profile for representing BOTL based metamodels as UML class diagrams and model variables as object diagrams. Therefore we extend the UML by two new types of tagged values that are defined in Table 3.1 and four new stereotypes that are listed in Table 3.2.

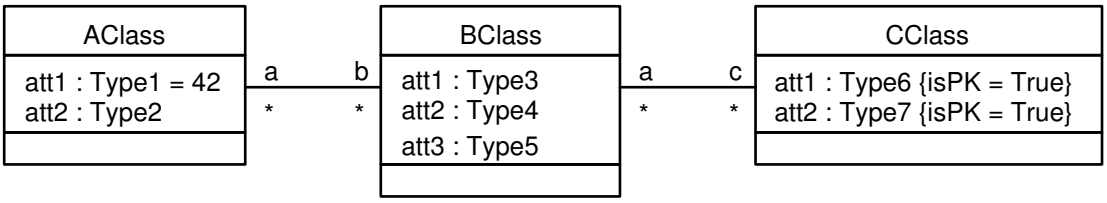

Figure 3.10: A sample BOTL meta model represented as a UML class diagram.

Figure 3.10 shows a sample BOTL metamodel in the presented UML notation that consists of three classes. As one can see only the class `CClass` does contain tags of the type `isPK`. Thereby it is determined that two instances (i.e. objects) of the class `CClass` must not have the same values for their key attributes `CClass.att1` *and* `CClass.att2`. The attribute `att1` of `AClass` has an initial value of 42 that intuitively can be interpreted as its default value in terms of BOTL.

Tag Definition	Tagged Element	Tag Element Type	Description
`isPK`	`Attribute`	`Boolean`	This tag indicates if the given attribute is part of a primary key for the instances of the class it belongs to. If a class has a number of primary key attributes this implies that there must not exist any two instances (objects) of this class that have pairwise identical values for all of these attributes. If no such tag is provided for an attribute it is assumed that the attribute is either not part of a primary key or that nothing is known about this attribute. The alternative graphical representation of attributes with the tag `isPK` is: The tag is left out and the regarding attributes are written in **bold** face.
`Term`	`Object`, `DataValue`	`Exp.`	A term as a tagged value for a `DataValue` determines how the value of this attribute can be retrieved during a transformation. In BOTL it conforms to the Terms t of object variables as defined in Definition 3.2.3. If `Term` serves as a tagged value for an object itself, it defines the expression that yields to the objects identity. It corresponds to the object identifier term in Definition 3.2.3. It may occur that the given object is of a class type that has primary keys in its class definition. In this case the object's term value must be an ordered tuple that consists of the term values of the attributes that are defined as primary keys. Alternatively the existence of primary key attributes can be indicated by writing the corresponding in **bold** face Figure 3.8 in Section 3.2 already provided an example for this representation style. For every data value or object identifier of an object variable that has no term tag a term tag with the value $\diamondsuit$ is assumed. Terms appear only in object variables where the data values of the UML object isn't relevant. Therefore an alternative graphical representation of the tagged values is allowed: All data values are be left out and replaced by the terms with all variables written in an italic style. For variables of a class type with primary keys a tuple of the key attribute names may be written as object identifiers.

Table 3.1: The UML tagged values of the BOTL profile

Stereotype	UML base class	Description
`Source Model Variable`	`Package`	A source model variable package indicates that the object model it contains represents a BOTL model variable as defined in 3.2.6 that represents the left side of a rule r, i.e. $r\|_{mv_0}$. Thus it must contain only object models, whereby every object represents a object variable that may contain terms (see Table 3.1). Since model variables must be coherent it is not mandatory to depict this package. If the two model variables of a rule can be distinguished by their position using the "arrow notation" for the rule the package may be omitted to improve the clarity of the representation. Throughout this work we usually omit the representation of this stereotyped package for a better readability.
`Target Model Variable`	`Package`	A target model variable package indicates that the object model it contains represents a BOTL model variable as defined in 3.2.6 that represents the right side of a rule r, i.e. $r\|_{mv_1}$. Like a source model variable package it must contain only object models. Again the representation of this package can be omitted if the two model variables of a rule can be distinguished otherwise.
`Rule`	`Package`	A rule package represents a BOTL model transformation rule as defined in Definition 3.2.7. Thus it must contain exactly one source model variable and one target model variable. As an alternative representation the two contained packages may be omitted and an arrow as shown in Figure 3.9 that points from the source to the target model variable is used to distinguish the two model variables.
`Rule Set`	`Package`	A rule set package represents a BOTL model transformation rule set as defined in Definition 3.2.8. As an alternative representation the representation of the rule packages and the rule package itself may be omitted if the structuring of the rules and their model variables is expressive enough.

Table 3.2: The UML stereotypes of the BOTL profile

For the representation of BOTL model variables we use a UML profile for object diagrams. Object diagrams are extended by two tagged values that are listed in Table 3.1 and the stereotype of Table 3.2.

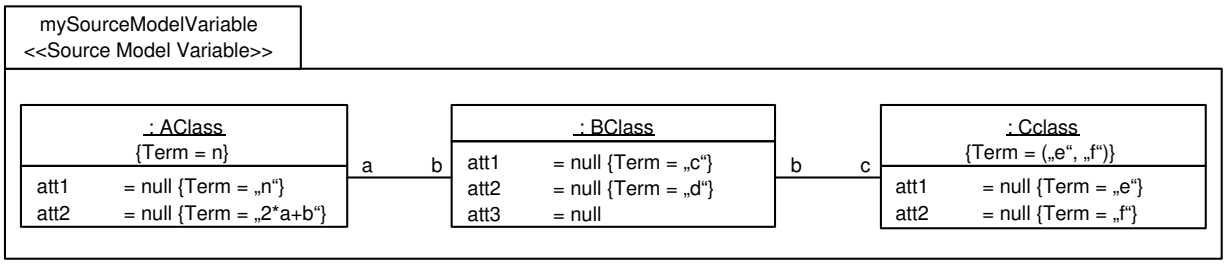

Figure 3.11: A sample BOTL model variable represented as an UML object diagram.

Figure 3.11 shows an object variable regarding the metamodel in Figure 3.10. There exists a term for every attribute except `BClass.att3`. Thus it is assumed that this attribute contains the term $\diamondsuit$.

The object variable of the type `AClass` has a term tag with the value *n* as its identity. This indicates that the identity of objects that are potentially generated from this object variable is determined by the value of the variable *n*. In this case this value can be also found in the attribute `AClass.att1`.

The object variable identity of the type `BClass` has no term attribute, thus $\diamondsuit$ is assumed here. This means that all objects that result from this object variable have uniquely generated identities.

The two attributes `att1` and `att2` of the class `CClass` are primary keys. Therefore the term of this object variable must be a tuple that consists of the terms of the primary key attributes as demanded in the description of Table 3.1.

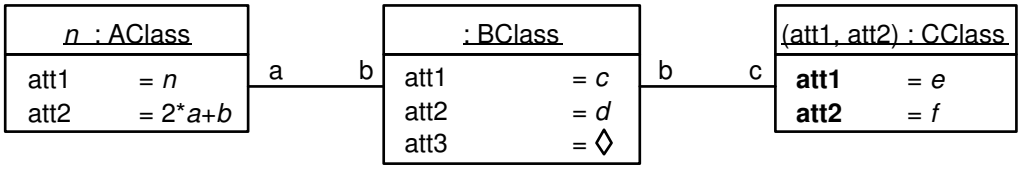

Figure 3.12: A more intuitive graphical representation of the model variable of Figure 3.11.

Figure 3.12 shows the same model variable as Figure 3.11 but now we use the alternative graphical representation. Since this representation seem much more intuitive one will usually prefer this notation. Therefore we also use this notation for model variables throughout this work.

Two distinguish model variables, rules, and rule sets one may group them into stereotyped packages. Table 3.2 lists the stereotypes that are used therefore. The use of this notation may be necessary, if one wants to specify a BOTL rule set with a standard UML tool. In general we will use the more readable '"arrow notation´" throughout this work.

3.4 Rule Set Application

In this section we define how BOTL transformation rules are applied using the basic formalization of object and class models and the definition of rule sets provided in the previous sections.

A model fragment is similar to a model (s. Def. 3.1.12), but it is not necessarily consistent in respect to the cardinalities of associations. Furthermore within model fragments attributes may be assigned the special value $\diamondsuit$ which may also be used within terms marking unset values.

Definition 3.4.1 (Model Fragment *mf*)
A *model fragment mf* regarding a metamodel *mm* is a tuple (OB, OA) that consists of

- an object allocation OB
- a set of object associations OA for that it holds:

$$\forall oa \in OA : \forall oae \in oa|_{OAE} : oae|_o \in OB \tag{3.18}$$

The predicate *consistent* holds if a model fragment is consistent to a given metamodel.

$$consistent(mf, mm) :\Leftrightarrow consistent(OB, mm) \wedge \forall oa \in OA : consistent(oa, OB)$$

❍

(3.18) corresponds to (3.15). I.e. a model fragment contains not necessarily a correct number of object associations for all class associations. According to (3.18) all existing object associations in OA are situated between objects o of the model fragment mf. The definition of object associations already ensures that the types of the objects are correct.

Definition 3.4.2 (Set of possible model fragments $\mathbb{MF}_{mm}$)
MF_{mm} is the set of all possible model fragments regarding a metamodel *mm*. ❍

Definition 3.4.3 (Model Fragment Match mfm_i)
A *model fragment match* mfm_i is a tuple $(mv, mf, match_{\mathrm{o}}, match_{\mathrm{a}})$ consisting of

- a model variable mv,
- a model fragment mf with $consistent(mf, mv|_{mm})$,
- a bijective mapping

  $$match_{\mathrm{o}} : mv|_{OVB} \rightarrow OB$$

  which maps object variables to object of the appropriate type:

  $$\forall ov \in mv|_{OVB} : match_{\mathrm{o}}(ov)|_{ot} = ov|_{otv} \tag{3.19}$$

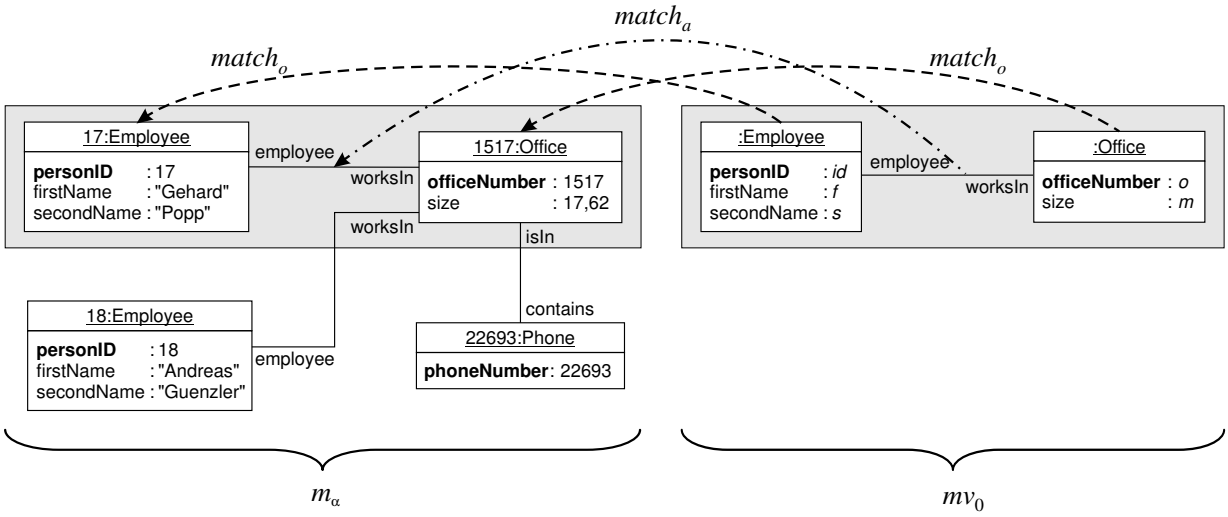

Figure 3.13: A model fragment match $(mv_0, mf, match_o, match_a)$

- a bijective mapping

  $match_a : mv|_{OVA} \rightarrow OA$

  which maps object variable associations to appropriate object associations:

$$\begin{aligned} &\forall ova \in OVA : \\ &\quad match_a(ova)|_{OAE} = \\ &\qquad \left\{(ovae|_{ae}, match_o(ovae|_{ov})) : ovae \in ova|_{OVAE}\right\} \\ &\wedge\ match_a(ova)|_{card} = ova|_{cardv} \end{aligned} \tag{3.20}$$

❍

Model fragment matches are needed to identify those parts of a given source model that match to the left hand side of a rule. They are also used to construct model fragments of the target model. In the example of Figure 3.13 one of two possible model fragment matches between the model m on the left side and the model variable mv_0 on the right side is shown.

$match_o$ takes an object variable and returns an object, so that every object variable of OVB is bijectively mapped to an object of OB of the same type (3.19). $match_a$ takes an object variable association and returns an object association, so that all object variable associations of OVA are bijectively mapped to an object association of OA (3.20) with the same association ends and the same cardinality. Furthermore for the result of $match_o$ and $match_a$ it has to hold that if an object variable and object variable association are connected also their matches are connected.

Theorem 3.4.1
All identifiers of object variables of a model fragment match mfm_i are assigned to pairwise different object identifier values. ❏

Proof: Theorem 3.4.1 holds trivially because the identifiers of all objects of an object allocation are pairwise different and a bijective mapping $match_o$ is postulated. ❏

Definition 3.4.4 (Model fragment match sequence $MFM(m,mv)$)
For a model m and a model variable mv $MFM(m,mv)$ is the set of all (finite) *model fragment match sequences mfm*. A model fragment match sequence *mfm* is a sequence of model fragment matches $(mfm_0,\ldots,mfm_n)$. Thereby it has to hold:

(i) $\forall i,j : i \neq j : mfm_i \neq mfm_j$

(ii) $\forall mfm_i$ with $i \in \{0,\ldots,n\} : mfm_i|mf|_{OB} \subseteq m|_{OB}$

(iii) $\forall mfm_i$ with $i \in \{0,\ldots,n\} : mfm_i|mf|_{OA} \subseteq m|_{OA}$

(iv) $mv = mfm_i|_{mv}$

(v) $\forall mfm_\tau : mfm_\tau$ complies to (i)- $(iv) \Rightarrow mfm_\tau \in mfm$

❍

(v) denotes that every *mfm* in $MFM(m,mv)$ does contain all possible "matches" of *mv* in *m*.

The number of model fragment matches to a given finite model and a given finite model variable is finite. For the application of a rule an arbitrary ordered sequence of model fragment matches is needed.

Example: In our running example there we can figure out one possible model fragment match

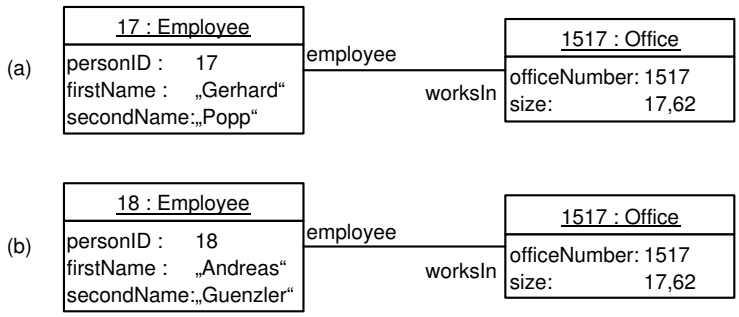

Figure 3.14: The model fragments (a) $mfm_0(r_0|_{mv_0})$ and (b) $mfm_1(r_0|_{mv_0})$

sequence $mfm = (mfm_0, mfm_1) \in MFM(m_\alpha, r_0|_{mv_1})$ of all (in our case two) possible model fragment matches in m_α and the model variable $r_0|_{mv_0}$. Figure 3.14 shows the model fragment matches mfm_0 and mfm_1. ❍

Definition 3.4.5 (Model Fragment Relation *mfr*)
A model fragment relation *mfr* is a relation between a model fragment match mfm_v with a model transformation rule r_u and a model fragment mf with $consistent(mf, r_u|_{mv_1}|_{mm})$:

$$mfr : (MFM \times \mathbb{R}_{mm_0,mm_1}) \times \mathbb{MF}_{mm_1}$$

Let

$$\begin{aligned}
mm_0 &= r_u|_{mv_0}|_{mm} \\
mm_1 &= r_u|_{mv_1}|_{mm} \\
m_0 &\in \mathbb{M}_{r_u|_{mv_0}|_{mm}} \text{ , arbitrary but fix} \\
m_1 &\in \mathbb{M}_{r_u|_{mv_1}|_{mm}} \text{ , arbitrary but fix} \\
MFM &= \{mfm : mfm \in MFM(m, r_u|_{mv_0}) \text{ with } m \in \mathbb{M}_{mm_0}, r_u \in \mathbb{R}_{mm_0,mm_1}\}
\end{aligned}$$

To be in a model fragment relation it is a prerequisite for mfm_v and r_u that it holds:

$$\exists mfm^0 \in MFM(m_0, r_u|_{mv_0}) \text{ with } \exists i \in \{0, \ldots, |mfm^0|\} : mfm^0_i = mfm_v \tag{3.21}$$

Let mf be chosen so that it holds:

$$\begin{aligned}
&\exists mfm^1_\mu : \exists mfm^1 \in MFM(m_1, r_u|_{mv_1}) \text{ with } mfm^1_\mu \in mfm^1 \\
\wedge\ &mfm^1_\mu|_{match_o}(r|_{mv_1}|_{OVB}) = mf|_{OVB} \\
\wedge\ &mfm^1_\mu|_{match_a}(r|_{mv_1}|_{OVA}) = mf|_{OVA}
\end{aligned} \tag{3.22}$$

Let GL^0_{id} be the set of equations that relate object identifiers of the source fragment to object variable identifiers of the left rule side. Thus GL^0_{id} contains all equations of the kind:

$$GL^0_{id} := \bigwedge_{\substack{ov \in r_u|_{mv_0}|_{OVB} \\ \wedge\ ov|_{oiv} \neq \diamondsuit\ \wedge\ ov|_{oiv} \neq \varepsilon}} mfm_v|_{match_o}(ov)|_{oi} = ov|_{oiv}$$

Let GL^0_v be the set of equations that relate the attribute values of all object variables of the left rule side to those of the source model fragment. Thus GL^0_v contains all equations of the kind:

$$GL^0_v := \bigwedge_{ov \in r_u|_{mv_0}|_{OVB}} \bigwedge_{att \in ov|_{VV}|_a \wedge ov.att \neq \diamondsuit} mfm_v|_{match_o}(ov).att = ov.att$$

Equations concerning identifiers resp. attribute values that are $\diamondsuit$ terms are ignored.

Let GL^1_{id} be the set of all equations that concern the set of object identifiers regarding all object variables of the right rule side. Thus GL^1_{id} contains all equations of the kind:

$$GL^1_{id} := \bigwedge_{ov \in r_u|_{mv_1}|_{OVB}} \begin{cases} pK(\{(a,v) \in mfm^1_\mu|_{match_o(ov)}|_V : \\ \quad \exists a \in ov|_{otv}|_{Keys}|_n\}) & \text{if } ov|_{oiv} = \varepsilon \wedge \\ & (\forall (a,t) \in ov|_{VV} \wedge \\ & \quad a \in ov|_{otv}|_{Keys}|_n : t \neq \diamondsuit) \\ ov|_{oiv} & \text{if } ov|_{oiv} \neq \diamondsuit \\ genID(u,v,genNum(ov, r_u|_{mv_1}|_{OVB})) & \text{otherwise} \end{cases}$$
$$= mfm^1_\mu|_{match_o}(ov)|_{oi}$$

Thereby let $genID(n_1,n_2,n_3)$ and $genNum(elem,set)$ be injective functions.

The function $genID(n_1,n_2,n_3)$ maps three numbers to an identifier in $\mathbb{ID}$ which will be not generated by any other identifier term.

The function $genNum(elem,set)$ yields to a number of $\mathbb{N}_0$ for an element $elem$ of a set set.

For object identifiers that would be identified as $\diamondsuit$ terms unique values are generated by using $genID$ and $genNum$.

Let GL^1_v be the set of equations that concern attribute values regarding all object variable of the right side of the rule. GL^1_v contains all equations of the kind:

$$GL^1_v := \bigwedge_{ov \in r_u|_{mv_1}|_{OVB}} \bigwedge_{att \in ov|_{VV}|_a} ov.att = mfm_v|_{match_o}(ov).att$$

Thereby equations for attribute values with $\diamondsuit$ terms are not ignored. These are replaced later with suitable default values.

It has to hold: The object identifiers and attributes of $mf|_{OB}$ are a solution for the system of equations that consist of the equations in $GL := GL^0_{id} \wedge GL^0_v \wedge GL^1_{id} \wedge GL^1_v$. ❍

Please note that (3.21) implies that the model variable of mfm_v is the same as the left rule side, i.e. $mfm_v|_{mv} = r_u|_{mv_0}$.

Example: In Figure 3.15 an example for a valid model fragment relation for the rule r_1 of our running example is given. There is a model fragment match mfm^0_i between the left rule variable mv_0 to a model fragment mf_0 that is indicated by the dashed pointers between the elements of the model variable and the model fragment. As required in Definition 3.4.5 (i) there exists also a match mfm^1_k for mv_1 that leads to another model fragment mf_1. The model variable mv_0 consists of the object variable $ov_{0,0}$, which is of the type Office and the object variable $ov_{0,1}$ of the type Phone. The model variable mv_1 consists only of one single object variable, which we denote by $ov_{1,0}$. According to the definition 3.4.5 we get the following equation system:

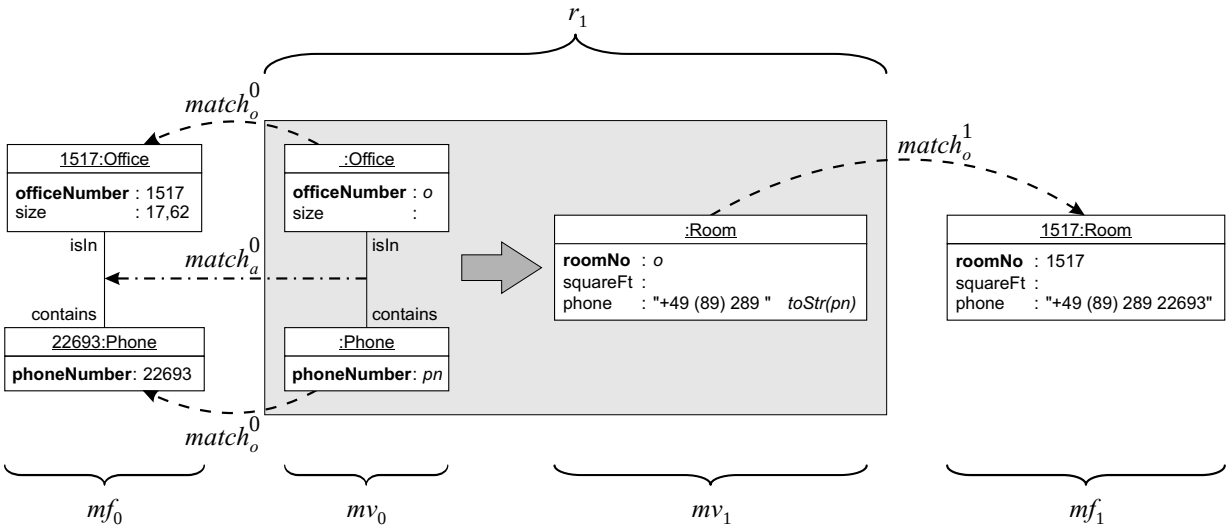

Figure 3.15: The rule r_1 with a model fragment match for the left side and a model fragment mf_1 where $mfr((mfm_i^0, r_1), mf_1)$ does hold.

$$
\begin{aligned}
GL_{id}^0 : (\quad & match_o^0(ov_{0,0})|oi && = \varepsilon \quad \text{not considered according to Def. 3.4.5)} \\
(\quad & match_o^0(ov_{0,1})|oi && = \varepsilon \quad \text{not considered according to Def. 3.4.5)} \\
GL_v^0 : \quad & match_o^0(ov_{0,0}).officeNumber && = o \\
& match_o^0(ov_{0,0}).size && = m \\
& match_o^0(ov_{0,1}).phoneNumber && = pn \\
GL_{id}^1 : \; & pK(\{(roomNo, match_o^1(ov_{1,0}).roomNo)\}) && = match_o^1(ov_{1,0})|oi \\
GL_v^1 : \quad & o && = match_o^1(ov_{1,0}).roomNo \\
& \diamondsuit && = match_o^1(ov_{1,0}).squareFt \\
& \text{"+49 (89) 289 "} \circ \text{toStr}(pn) && = match_o^1(ov_{1,0}).phone
\end{aligned}
$$

We can derive a solution for this equational system that yields to the identifier and attribute values for the generated objects in the target model for a given source fragement:

$$
\begin{aligned}
match_o^1(ov_{1,0})|oi &= pK(\{(roomNo, match_o^0(ov_{0,0}).officeNumber\}) \\
match_o^1(ov_{1,0}).roomNo &= match_o^0(ov_{0,0}).officeNumber \\
match_o^1(ov_{1,0}).squareFt &= \diamondsuit \\
match_o^1(ov_{1,0}).phone &= \text{"+49 (89) 289 "} \circ \text{toStr}(match_o^0(ov_{0,1}).phoneNumber)
\end{aligned}
$$

For the left hand model fragment shown in Figure 3.15 we get:

$$\begin{aligned} match^1_o(ov_{1,0})|oi &= pK(\{(roomNo, 1517\}) \\ match^1_o(ov_{1,0}).roomNo &= 1517 \\ match^1_o(ov_{1,0}).squareFt &= \diamondsuit \\ match^1_o(ov_{1,0}).phone &= \text{"+49 (89) 289 "} \circ \text{toStr(22693)} = \text{"+49 (89) 289 22693"} \end{aligned}$$

As one can see this is a solvable equational system with valid solutions for all variables. Thus $mfr((mfm^0_i, r_1), mf_1)$ does hold.

In general not all systems of equations can be resolved as easy as in this example. The example already shows that data types like strings together with the concatenation operator "$\circ$" may occur that do not form a mathematical group. Hence a useful tool support for the presented approach must provide the ability to plug-in user defined solution strategies. The problem becomes clear when imagining the reverse transformation from beta to alpha: now the user must provide a strategy how to extract the first and the second name from a single string name.

There may be several (or none) model fragments mf^l_1 that are in the relation $mfr((mfm^0_i, r_1), mf^l_1)$ for given mfm^0_i and r_1. Those relations that hold only for one model fragment mf_1 are of special interest, because therefore attribute values for model fragments can be computed deterministically. ❍

Lemma 3.4.1
It holds for $i \neq j$: $mfm^1_\mu|_{match_o}(ov_i)|_{oi} \neq mfm^1_\mu|_{match_o}(ov_j)|_{oi}$ ❑

Proof Sketch: Lemma 3.4.1 concludes directly from Definition 3.4.1, Definition 3.4.3 (3.19), and Definition 3.4.5 (3.22) because the objects of the target model are an object allocation. ❑

Lemma 3.4.2
For every object o of a model fragment mf that is in a model fragment relation $mfr : (mfm_v, r_u) \times mf$ with an arbitrary but valid (according to Definition 3.4.5) mfm_v and r_u *it*holds that:

$\forall(o, AE : o|_{ot} \in AE|_c, ae \in AE)$:
$\exists ov \in r_u|_{mv_1}|_{OVB}$ with $(i)\ o|_{ot} = ov|_{otv} \wedge$

$$(ii) \sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} = \sum_{\substack{oae:oae=(ae,o) \\ \wedge oae \in oa|_{OAE} \\ \wedge oa|_{OAT}=AE}} oa|_{card}$$

I.e. the sum of all cardinalities of all outgoing associations of the same *type*of an object o is equal to the sum of cardinalities of the corresponding association variables that go out of the object variable from which o might originate (s. example in Fig. 3.16). ❑

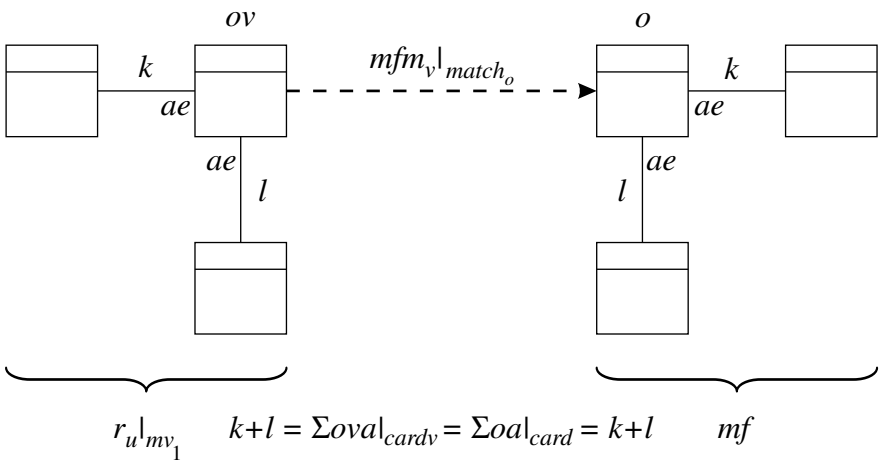

Figure 3.16: Sum of cardinalities of outgoing association

Proof Sketch: The proof arises from the properties (i) and (ii):

(i) because of Definition 3.4.5 (3.22) and Definition 3.4.3 (3.19).

(ii) because of Definition 3.4.3 (3.20) Definition 3.4.3 (3.19). ❑

Definition 3.4.6 (Model Fragment Transformation $mft(mfm_i, r_j)$)
If a model fragment relation *mfr* is a total function, then we call it a *model fragment transformation mft* of a model fragment *mf* by a rule r_j.

$$mft(mfm_i, r_j) \mapsto \begin{cases} mfr(mfm_i, r_j) & \text{if } mfr \text{ is a total function} \\ \perp & \text{otherwise} \end{cases} \tag{3.23}$$

❍

Example: A model fragment transformation returns a new model fragment whose structure is determined by the right side of the applied rule. Since all the values of the model frag-

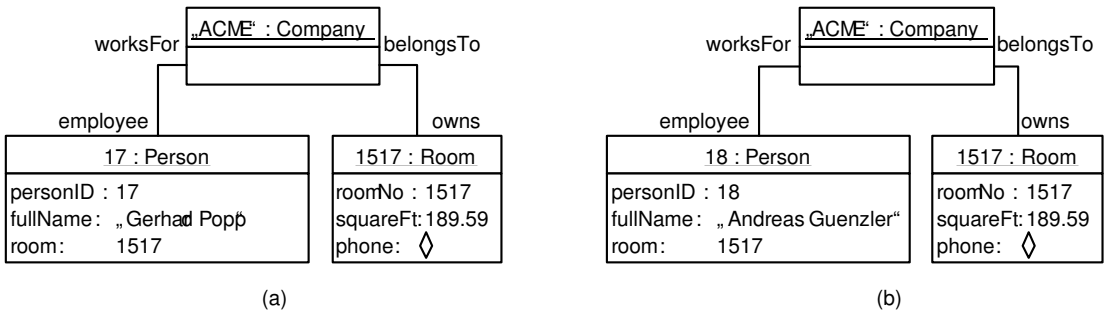

Figure 3.17: The created model fragments (a) $gmf_{0,0}$ and (b) $gmf_{0,1}$

ment relation in our running example can be derived deterministically it is a model fragment transformation *mft* with respect to Definition 3.4.6. The generated model fragment $gmf_{0,0} := mft(mfm_0, r_0)$ is depicted in Figure 3.17 (a). We call the newly generated model fragment $gmf_{0,0} := mft(mfm_0, r_0)$, since it is the first model fragment that was generated during the application of the first rule Likewise we get the second generated model fragment $gmf_{0,1} := mft(mfm_1, r_0)$ that is depicted in Figure 3.17 (b). ❍

A model fragment transformation transforms one match of the source model into exactly one model fragment of the target model. To transform a complete model all matches are transformed and resulting fragments are merged. Therefore in the following a predicate *mergeable* indicating the merge-ability of two object allocations and a merge operator operating on model fragments are defined.

Definition 3.4.7 ($mergeable(OB^0, OB^1)$**)**

$$\begin{aligned} mergeable(OB^0, OB^1) = (&\nexists o_0 \in OB^0, o_1 \in OB^1 : o_0|_{oi} = o_1|_{oi} \\ &\wedge ((\exists (a_0, v_0) \in o_0|_V, (a_1, v_1) \in o_1|_V \\ &\quad \text{with } a_0 = a_1 \wedge v_0 \neq v_1 \wedge v_0 \neq \diamondsuit \wedge v_1 \neq \diamondsuit) \\ &\vee o_0|_{ot} \neq o_1|_{ot})) \end{aligned}$$

❍

Two object allocations are *mergeable*, iff there exists no object in both allocations with an identical identificator but with an contradictory object type or contradictory attribute values.

Definition 3.4.8 ($OBmerge(OB^0, OB^1)$**)**
Let *OBmerge* be a strict mapping of two object allocations OB^0 and OB^1 regarding the same metamodel to another object allocation.

$$OBmerge(OB^0, OB^1) = \begin{cases} OB^+ & \text{iff } mergeable(OB^0, OB^1) \\ \perp & \text{otherwise} \end{cases}$$

Thereby OB^+ is the set of objects for that holds:

$\forall o_0 \in OB^0$ with $o_0|_{oi} \notin OB^1|_{oi} : o_0 \in OB^+$

$\forall o_1 \in OB^1$ with $o_1|_{oi} \notin OB^0|_{oi} : o_1 \in OB^+$

$\forall o_0, o_1$ with $o_0 \in OB^0 \wedge o_1 \in OB^1$ with $o_0|_{oi} = o_1|_{oi}$:

$\exists_1 o^+ \in OB^+ : o^+|_{oi} = o_0|_{oi} \wedge o^+|_{oi} = o_1|_{oi}$

$\wedge\, o^+|_{ot} = o_0|_{ot} \wedge o^+|_{ot} = o_1|_{ot}$

$$\wedge\, o^+|_V = \Big\{ (a,v) : \ (a,v_0) \in o_0|_V \wedge (a,v_1) \in o_1|_V \wedge v = \begin{cases} v_0 & \text{for } v_0 \neq \diamondsuit \\ v_1 & \text{otherwise} \end{cases} \Big\}$$

❍

The result of *OBmerge* is an object allocation containing all objects of the source allocations if these are *mergeable*. The attributes of the resulting objects contain the values of their origins but $\diamondsuit$ values may be overwritten by concrete values. If the two object allocations are not mergeable (e.g. in case of conflicting attribute values) $\perp$ is returned. Please note that $\perp$ as an argument of *OBMerge* always yields to $\perp$, since *OBMerge* is a strict mapping.

Lemma 3.4.3
For $mergeable(OB^0, OB^1)$ it holds that:

$$OBmerge(OB^0, OB^1)|_{oi} = OB^0|_{oi} \cup OB^1|_{oi}$$

❑

Proof Sketch: Lemma 3.4.3 results direct from Definition 3.4.8. ❑

Definition 3.4.9 (Merge $mf_0 \cup_{\mathfrak{m}} mf_1$)
$\cup_{\mathfrak{m}}$ is a strict mapping:

$\cup_{\mathfrak{m}} : MF_{mm} \times MF_{mm} \to MF_{mm}$

$$mf_0 \cup_{\mathfrak{m}} mf_1 \mapsto \begin{cases} \perp & \text{for } mf_i = \perp \text{ with } i \in \{0,1\} \\ & \vee \neg mergeable(mf_0|_{OB}, mf_1|_{OB}) \\ (OBmerge(mf_0|_{OB}, & \\ \quad mf_1|_{OB}), OA^+) & \text{otherwise} \end{cases}$$

With

$$OA^{+} = (mf_0|_{OA} \cup mf_1|_{OA}) \setminus \{\, (OAT, card, OAE) : \exists oa_k \in mf_k \text{ with } k \in \{0,1\} : \\ OAT = oa_k|_{OAT} \\ OAE = oa_k|_{OAE} \\ oa_0|_{card} \neq oa_1|_{card} \\ card = \min\{oa_0|_{card}, oa_1|_{card}\} \,\}$$

❍

Merging $\perp$ with any fragment results in $\perp$. Two objects with the same *oi* are merged to one object. Corresponding attributes must have the same value or at least one value is $\diamondsuit$ which will be overwritten. Otherwise the resulting model fragment is $\perp$. All object associations are preserved. OA^{+} is the union of all object associations of the model fragments mf_0 and mf_1, but if there are associations in mf_0 and mf_1 that are equal except of their cardinalities *card*, then only that one with the higher cardinality is in OA^{+}. Thus the cardinality of the resulting object associations is the maximum cardinality of the corresponding object associations in the source model fragments.

We use the following abbreviation:

$$\bigcup_{\substack{\mathfrak{m} \\ i=\{0,\ldots,n\}}} mf_i := mf_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} mf_n$$

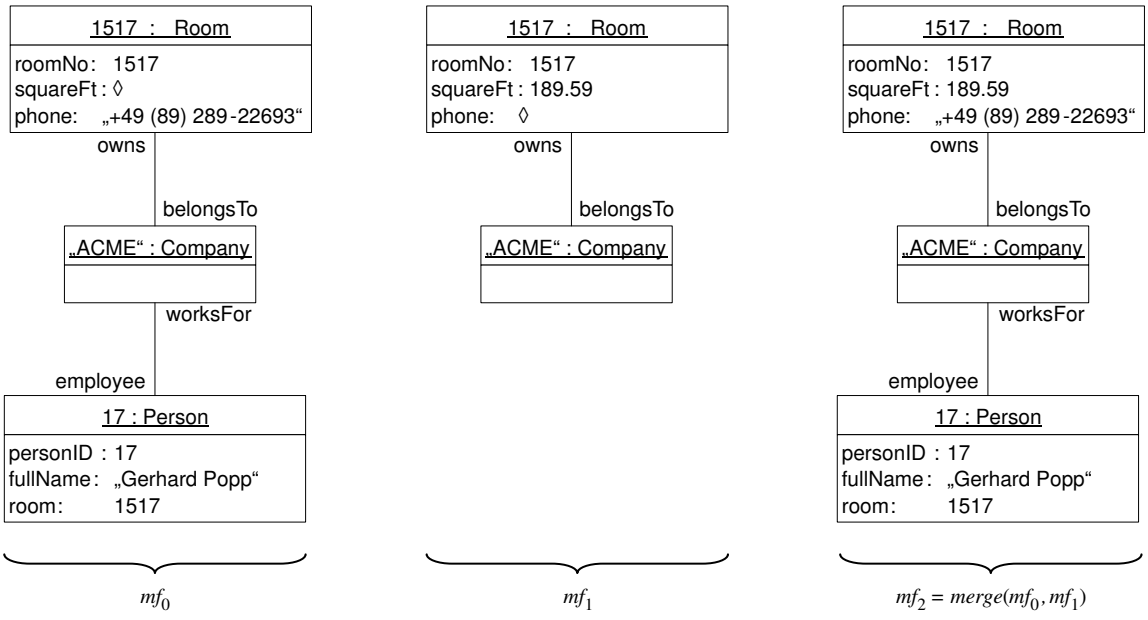

Figure 3.18: Two *mergeable* model fragments and the result of $\cup_{\mathfrak{m}}$

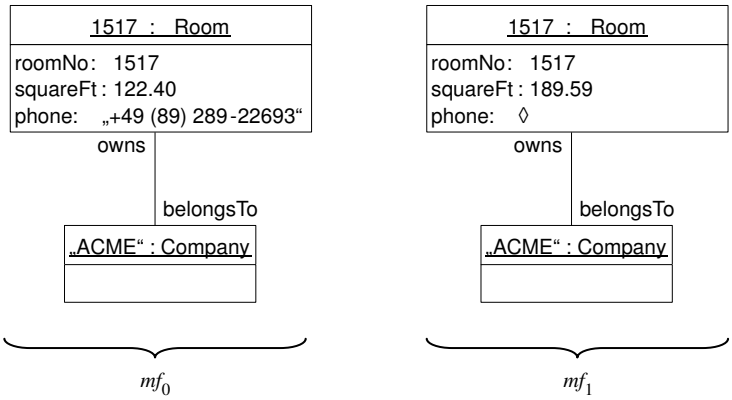

Figure 3.19: Two model fragments which are not *mergeable*

Example: In Figure 3.18 three model fragments mf_0, mf_1, and mf_2 are shown. The merge of model fragments mf_0 and mf_1 result in the model fragment mf_2. Figure 3.19 contains two other model fragments mf_0 and mf_1, which are not mergeable because the attribute squareFt gets assigned two conflicting values. ❍

For a better readability and a more convenient handling we conclude some key issues of Definition 3.4.9 in the following Lemma:

Lemma 3.4.4 (Merge-Lemma)
For $mf_2 = mf_0 \cup_{\mathfrak{m}} mf_1$ it holds that:

- (i) $mf_2|_{OA}$ contains all object associations that occur in $mf_0|_{OA}$ or $mf_1|_{OA}$. If object associations occur in both model fragments then only these with the bigger cardinality *card* are included in $mf_2|_{OA}$.
- (ii) $mf_2|_{OB}|_{oi} = mf_0|_{OB}|_{oi} \cup mf_1|_{OB}|_{oi}$
- (iii) $mf_2|_{OA} \subseteq mf_0|_{OA} \cup mf_1|_{OA}$

❑

Proof Sketch: The proof of Lemma 3.4.4 is directly evident from Definition 3.4.9, Lemma 3.4.1 and Lemma 3.4.3. ❑

The following theorems state some interesting properties of the relation *mergable* and the $\cup_{\mathfrak{m}}$ operator: *mergeable* as well as $\cup_{\mathfrak{m}}$ are commutative and associative. The proof is based upon the commutativity and associativity of $\cup$.

Theorem 3.4.2
The relation *mergeable* is commutative.

$mergeable(OB^0, OB^1) = mergeable(OB^1, OB^0)$

❏

Proof Sketch: Theorem 3.4.2 can be proved easily by substitution into the definition of *mergeable* (Def. 3.4.7). ❏

Theorem 3.4.3
The strict mapping *OBmerge* is commutative.

$$OBmerge(OB^0, OB^1) = OBmerge(OB^1, OB^0)$$

❏

Proof Sketch: Theorem 3.4.3 can be proved easily by substitution in the definition of OBmerge (Def. 3.4.8). ❏

Lemma 3.4.5
For arbitrary $OB'_0 \subseteq OB_0, OB'_1 \subseteq OB_1$ it holds:

(i) $mergeable(OB_0, OB_1) \Rightarrow mergeable(OB'_0, OB'_1)$

(ii) $\neg mergeable(OB_0, OB_1) \Rightarrow \exists o_0 \in OB_0, o_1 \in OB_1$ with $\neg mergeable(\{o_0\}, \{o_1\})$

(iii) $\neg mergeable(OB'_0, OB'_1) \Rightarrow \neg mergeable(OB_0, OB_1)$

❏

Proof Sketch: (i) and (ii) are evident directly from Definition 3.4.7. According to modus tollens (iii) follows directly from (i). ❏

Lemma 3.4.6
For arbitrary $OB'_0 \subseteq OB_0, OB'_1 \subseteq OB_1$ it holds:

$$\begin{aligned} & OBmerge(OB'_0, OB'_1) = \bot \\ \Rightarrow\ & OBmerge(OB_0, OB_1) = \bot \end{aligned}$$

❏

Proof:

$$\begin{array}{ll} & OBmerge(OB'_0, OB'_1) = \bot \\ \Rightarrow & \neg mergeable(OB'_0, OB'_1) \\ \overset{\text{Lemma3.4.5}(iii)}{\Rightarrow} & \neg mergeable(OB_0, OB_1) \\ \Rightarrow & OBmerge(OB_0, OB_1) = \bot \end{array}$$

❑

The following lemma states that object identities don't disappear within a *OBmerge* operation.

Lemma 3.4.7

$$\begin{array}{l} o|_{oi} \in (OB_0|_{oi} \cup OB_1|_{oi}) \wedge mergeable(OB_0, OB_1) \\ \Rightarrow o|_{oi} \in OBmerge(OB_0, OB_1)|_{oi} \end{array}$$

❑

Proof Sketch: For the three possible cases

(i) $o|_{oi} \in OB_0|_{oi} \wedge o|_{oi} \notin OB_1|_{oi}$

(ii) $o|_{oi} \notin OB_0|_{oi} \wedge o|_{oi} \in OB_1|_{oi}$

(iii) $o|_{oi} \in OB_0|_{oi} \wedge o|_{oi} \in OB_1|_{oi}$

the statement is evident directly from the three corresponding cases of Definition 3.4.8. ❑

Lemma 3.4.8
It holds for $OBmerge(OB^0, OBmerge(OB^1, OB^2)) \neq \bot$ that:

(i) $mergeable(OB^0, OB^1)$

(ii) $mergeable(OB^0, OB^2)$

(iii) $mergeable(OB^1, OB^2)$

❑

Proof: (iii) has to hold obviously because *OBmerge* is strict. $(i) \Leftrightarrow (ii)$ because *OBmerge* is commutative, i.e. $OBmerge(OB^0, OB^1) = OBmerge(OB^1, OB^0)$

w.l.o.g. we prove (ii):

Proof by contradiction:

$$\begin{aligned}
& \neg mergeable(OB^0, OB^2) \\
\overset{\text{Lemma 3.4.5}}{\Rightarrow} \; & \exists o', o'' \text{ with } \neg mergeable(\{o'\},\{o''\}) \wedge o' \in OB^0 \wedge o'' \in OB^2 \\
\Rightarrow \; & o'|_{oi} = o''|_{oi}
\end{aligned}$$

Case 1:

$$\begin{aligned}
& o'|_{ot} \neq o''|_{ot} \\
\overset{\text{Lemma 3.4.6}}{\Rightarrow} \; & \exists o_m \in OBmerge(OB^1,\{o''\}) \text{ with } o_m|_{oi} = o''|_{oi} \wedge o_m|_{ot} = o''|_{ot} \\
& \qquad \wedge o_m|_{oi} = o'|_{oi} \wedge o_m|_{ot} = o'|_{ot} \\
\Rightarrow \; & \neg mergeable(\{o'\},\{o_m\}) \\
\Rightarrow \; & OBmerge(\{o'\},\{o''\}) = \bot \\
\overset{\text{Lemma 3.4.6}}{\Rightarrow} \; & OBmerge(OB^0, OBmerge(OB^1, OB^2)) = \bot \\
& \lightning \text{ (Contradiction)}
\end{aligned}$$

Case 2:

$$\begin{aligned}
& \exists (a,v') \in o'|_V \wedge \exists (a,v'') \in o''|_V \text{ with } v' \neq v'' \wedge v' \neq \Diamond \wedge v'' \neq \Diamond \\
\overset{\text{Lemma 3.4.5}}{\Rightarrow} \; & \exists o_m \in OBmerge(OB^1,\{o''\}) \text{ with } o_m|_{oi} = o''|_{oi} \wedge (a,v'') \in o_m|_V \\
\Rightarrow \; & OBmerge(\{o'\},\{o_m\}) = \bot \\
\overset{\text{Lemma 3.4.6}}{\Rightarrow} \; & OBmerge(OB^0, OBmerge(OB^1, OB^2)) = \bot \\
& \lightning \text{ (Contradiction)}
\end{aligned}$$

$\Rightarrow (ii), (i)$

❑

Theorem 3.4.4
The mapping *OBMerge* is associative.

$$OBmerge(OB^0, OBmerge(OB^1, OB^2)) = OBmerge(OBmerge(OB^0, OB^1), OB^2)$$

❑

Proof: Let $l.s. := OBmerge(OB^0, OBmerge(OB^1, OB^2))$ and
$r.s. := OBmerge(OBmerge(OB^0, OB^1), OB^2)$

According to Lemma 3.4.8 it holds: $r.s. = \bot \wedge l.s. = \bot$ or $r.s. \neq \bot \wedge l.s. \neq \bot$

Case 1: $r.s. = \bot \wedge l.s. = \bot$
$\Rightarrow l.s. = r.s.$

Case 2: $r.s. \neq \bot \wedge l.s. \neq \bot$

Proof by contradiction: It holds (I) or (II)

$$l.s.|_{OB}|_{oi} \neq r.s.|_{OB}|_{oi} \tag{I}$$

$$\exists o' \in l.s.|_{OB}, o'' \in r.s.|_{OB} \text{ with } o'|_{oi} = o''|_{oi} \wedge (o'|_{ot} \neq o''|_{ot} \vee o'|_V \neq o''|_V) \tag{II}$$

Case I: $\exists o$ with $o \in l.s.|_{OB} \cup r.s.|_{OB} \wedge o|_{oi} \notin l.s.|_{OB}|_{oi} \cap r.s.|_{OB}|_{oi}$

$\overset{\text{Lemma 3.4.7}}{\Rightarrow}$ $\lightning$ (Contradiction)

Case II: according to Definition 3.4.8 it holds that:

$$o'|_{ot} = o''|_{ot} \wedge o'|_V|_a = o''|_V|_a$$
$$\Rightarrow \exists (a, v') \in o'|_V \wedge \exists (a, v'') \in o''|_V \text{ with } v' \neq v''$$

according to Lemma 3.4.5 it holds that:

$$mergeable(\{o'\}, \{o''\})$$
$$\overset{v' \neq v''}{\Rightarrow} (v' = \diamondsuit \wedge v'' \neq \diamondsuit) \vee (v' \neq \diamondsuit \wedge v'' = \diamondsuit)$$
$$\overset{\text{Def. OBmerge}}{\Rightarrow} \lightning \text{ (Contradiction)}$$

❑

Theorem 3.4.5
The $\cup_{\mathfrak{m}}$ operation is commutative, i.e. $mf_0 \cup_{\mathfrak{m}} mf_1 = mf_1 \cup_{\mathfrak{m}} mf_0$. ❑

Proof: *OBmerge* is commutative according to Theorem 3.4.3. Thus it can easily be shown by substitution that Theorem 3.4.5 does hold. ❑

Theorem 3.4.6
The $\cup_{\mathfrak{m}}$ operation is associative, i.e.

$$(mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2 = mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2)$$

❑

Proof: For the proof of Theorem 3.4.6 three cases can be differentiated, whereby

$l.s. := (mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2$
$r.s. := mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2)$

Case 1: $l.s. = \bot \wedge r.s. = \bot \Rightarrow l.s. = r.s.$

Case 2: w.l.o.g. (according to Theorem 3.4.5) $l.s. = \bot \wedge r.s. \neq \bot$

$$\begin{aligned}\bot \neq r.s.|_{OB} &= (mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2))|_{OB} \\ &= OBmerge(mf_0|_{OB}, OBmerge(mf_1|_{OB}, mf_2|_{OB})) \\ &= OBmerge(OBmerge(mf_0|_{OB}, mf_1|_{OB}), mf_2|_{OB}) \\ &= ((mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2)|_{OB} \\ &= l.s.|_{OB}\end{aligned}$$

This contradicts to the assumption $l.s. = \bot$.

Case 3: $l.s. \neq \bot \wedge r.s. \neq \bot$

$$\begin{aligned}r.s.|_{OB} &= (mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2))|_{OB} \\ &= OBmerge(mf_0|_{OB}, OBmerge(mf_1|_{OB}, mf_2|_{OB})) \\ &= OBmerge(OBmerge(mf_0|_{OB}, mf_1|_{OB}), mf_2|_{OB}) \\ &= ((mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2)|_{OB} \\ &= l.s.|_{OB}\end{aligned}$$

According to Definition 3.4.9 $r.s.|_{OA}$ resp. $l.s.|_{OA}$ contains all object associations that occur in mf_0, mf_1, or mf_2. If an object association occurs in several model fragments mf_i than exactly those with the maximum cardinality *card* are contained in $l.s.|_{OA}$ (associativity of the binary *max* resp. *min* function).

Thus it holds: $l.s. = r.s.$ ❑

Theorem 3.4.7
For all permutations *perm* of a sequence $i = 1, ..., n$ it holds:

$$((\cdots(mf_1 \cup_{\mathfrak{m}} mf_2) \cup_{\mathfrak{m}} \cdots) \cup_{\mathfrak{m}} mf_{n-1}) \cup_{\mathfrak{m}} mf_n = ((\cdots(mf_{perm(1)} \cup_{\mathfrak{m}} mf_{perm(2)}) \cup_{\mathfrak{m}} \cdots) \cup_{\mathfrak{m}} mf_{perm(n-1)}) \cup_{\mathfrak{m}} mf_{perm(n)}$$

❑

Proof: Theorem 3.4.7 does hold because $\cup_{\mathfrak{m}}$ is commutative and associative. ❑

Definition 3.4.10 (Rule Application ($apply(m, r_i, mf_0)$))
A *rule application* is a mapping

$$apply : \mathbb{M}_{mm_0} \times \mathbb{R}_{mm_0,mm_1} \rightarrow \mathbb{MF}_{mm_1}$$
$$apply(m, r_i, mf_0) \mapsto mf_{1,|mfm|}$$

where *mfm* is an arbitrary but fix (finite) model fragment match sequence with

$mfm \in MFM(m, r_i|_{mv_0})$

and

$$mf_{1,j} = \begin{cases} mf_0 & \text{for } j = 0 \\ (mf_{1,j-1} \cup_{\mathfrak{m}} mft(mfm_{j-1}, r_i)) & \text{for } 1 \leq j \leq |mfm| \end{cases}$$

❍

Theorem 3.4.8
The result of a rule application (Definition 3.4.10) is invariant according to all possible model fragment match sequences regarding a model *m* and a model variable *mv* (Definition 3.4.4). ❑

Proof Sketch: Theorem 3.4.8 concludes directly from Theorem 3.4.7. ❑

Theorem 3.4.8 states an important property of BOTL: within a rule application we extend the target model by finding matches for a rule's left side model variable in the source model, creating new model fragments for the new target model and merging them one after the other into the target model. According to Theorem 3.4.8 the (chronological) order in which matches are found does not matter. Consequently the order of the merges of new fragments doesn't matter, too. Thus a rule application always yields to a deterministic result, independent of the chosen pattern matching strategy that is used to find matches in the source model.

Definition 3.4.11 (Rule Set Application (*transform*(m,r)))
A *rule set application* is a mapping

$transform : \mathbb{M}_{mm_0} \times \mathbb{RW}_{mm_0,mm_1} \rightarrow \mathbb{MF}_{mm_1}$
$transform(m,r) \mapsto mf$

so that it does hold

$mf_0 = (\emptyset, \emptyset)$
$mf_i = apply(m, r_{i-1}, mf_{i-1})$, for $i \in \{1, \ldots, |r|\}$
$mf = mf_{|r|}$

Whereby all $\diamondsuit$ values in *mf* are replaced by appropriate default values. ❍

Theorem 3.4.9
The result of a rule set application is invariant with respect to the ordering of the rules of the given rule set *r*. ❑

Proof Sketch: Theorem 3.4.9 is a direct consequence of Theorem 3.4.7. ❏

Theorem 3.4.9 states another important property of BOTL. The order in which rules of a rule set are applied does not influence the result of a rule set application. This means that every rule is independent of all other rules which leads to better modularity of rule sets.

Theorem 3.4.10
Generally it holds:

$$transform(m,r) = \bigcup_{\substack{\mathfrak{m} \\ r_i : r_i \in r}} apply(m, r_i, (\emptyset, \emptyset))$$

❏

Proof:

$$transform(m,r) \overset{\text{Def. 3.4.11}}{=} apply(m, r_{|r|-1}, apply(\ldots, apply(m, r_0, (\emptyset,\emptyset))\ldots)) \overset{\text{Def. 3.4.10, Thm. 3.4.5}}{=} \bigcup_{\substack{\mathfrak{m} \\ r_i : r_i \in r}} apply(m, r_i, (\emptyset,\emptyset))$$

❏

Example: In terms of our formalism a model transformation is a rule set application as defined in Definition 3.4.11. Accordingly the result of the transformation in our running example can be obtained from the function $transform(m_\alpha, r)$. Inserting the example's values leads to

$$\begin{aligned} transform(m_\alpha, r) &\mapsto mf_2 \quad \text{, with} \\ mf_2 &= apply(m_\alpha, r_1, apply(m_\alpha, r_0, (\emptyset,\emptyset))) \end{aligned}$$

$$transform(m_\alpha, r) \mapsto mf_2 = apply(m_\alpha, r_1, apply(m_\alpha, r_0, (\emptyset,\emptyset)))$$

The model fragment match sequence mfm from our running example can be used to resolve the inner *apply* operator according to Definition 3.4.10:

$$\begin{aligned} apply(m_\alpha, r_0, (\emptyset,\emptyset)) &= mf_1 \quad \text{, with} \\ mf_1 &= mf_{1,2} \\ &= mf_{1,1} \cup_{\mathfrak{m}} mft(mfm_1, r_0) \\ &= (mf_{1,0} \cup_{\mathfrak{m}} mft(mfm_0, r_0)) \cup_{\mathfrak{m}} mft(mfm_1, r_0) \\ &= (mf_0 \cup_{\mathfrak{m}} mft(mfm_0, r_0)) \cup_{\mathfrak{m}} mft(mfm_1, r_0) \\ apply(m_\alpha, r_0, (\emptyset,\emptyset)) &= mf_{1,2} = mf_{1,1} \cup_{\mathfrak{m}} mft(mfm_1, r_0) \\ &= (mf_0 \cup_{\mathfrak{m}} mft(mfm_0, r_0)) \cup_{\mathfrak{m}} mft(mfm_1, r_0) \end{aligned}$$

When the first rule r_0 is applied it holds that $mf_0 = (\emptyset, \emptyset)$. Thus

$$mf_0 \cup_{\mathfrak{m}} mft(mfm_0, r_0) = (\emptyset, \emptyset) \cup_{\mathfrak{m}} gmf_{0,0} = gmf_{0,0}$$

I.e. the model fragment $gmf_{0,0}$ is merged into an empty model fragment. Resolving the *apply* function according to 3.4.9 we retrieve gmf_0 (s. Fig. 3.20): This statement is rather trivial, since the first model fragment $gmf_{0,0}$ is merged simply into an empty model fragment. Resolving the *apply* function according to Definition 3.4.9 we retrieve the model fragment gmf_0 shown in Figure 3.20:

$$apply(m_\alpha, r_0, (\emptyset, \emptyset)) = gmf_{0,0} \cup_{\mathfrak{m}} gmf_{0,1} = gmf_0$$

The value for the attribute phone will be generated by the application of the second rule. Ac-

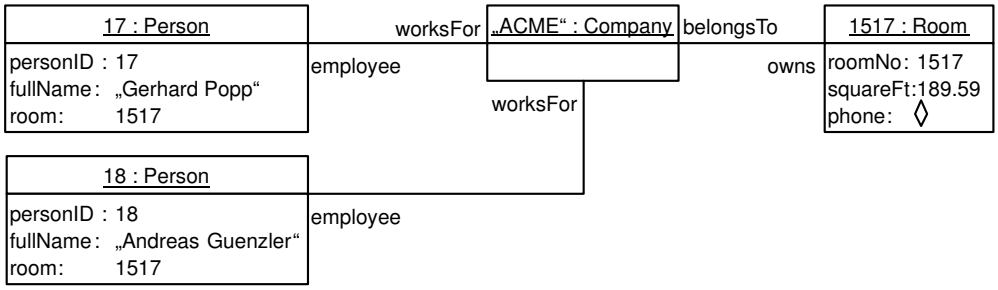

Figure 3.20: The result of $apply(m_\alpha, r_0, (\emptyset, \emptyset)) = gmf_0$: The Person and Room objects are already created, but there is no value for the phone attribute yet.

cording to Theorem 3.4.7 we could also permute the model fragment matches mfm_0 and mfm_1 and would still get the same result.

The application of the second rule r_1 is performed analogous, but it starts with gmf_0. Since the merge operator ensures, that already created objects are merged with those that are newly created, the phone attribute is set correctly in all existing Room objects. Theorem 3.21 ensures that r_0 could be exchanged with r_1. The result of the transformation is shown in Figure 2.4 at page 8.

❍

So far we defined a model for classes, objects and rule sets and determined how to apply rule sets. However the application of a rule set may still yield to $\perp$ as the result, which indicates that the transformation process could not be performed successfully. In the next sections we provide techniques that allow to determine weather a rule set can be applied successfully with respect to two given source and target metamodels. Further aspects of interest are meta model conformance and bijectivity.

4 Properties of BOTL Rule Sets

In the last section we have introduced the fundamentals of the BOTL formalism. With these fundamental concepts transformations of models can already be specified. But up to now there is no methodology for creating reasonable rules and rule sets for different purposes. So in the next three sections some techniques are discussed that allow to determine:

- if a rule or a rule set is *applicable*, that means the result of an application is always different to $\perp$ (c.f. Section 4.1), and
- if a rule or a rule set is *metamodel conform*, that means the resulting model fragment is also a model regarding the target metamodel (c.f. Section 4.2.

Most of these verification techniques are only sufficient but not always necessary postulations. Nevertheless they are proven by argumenting about the according metamodels and the model variables.

Since one of the purposes of BOTL was to describe transformation of data between different CASE tools also some kind of bijectivity of the transformations has to be discussed. In Section 4.3 the meaning of bijectivity in the context of BOTL is defined.

4.1 Applicability

Applicability is the first of three important properties of rules and rule sets. We tried to make the verification technique for that property simple enough to be understood and applied easily whereby they also have to be general enough to be usable.

Definition 4.1.1 (Applicability of a Rule Set)
A rule set r is *applicable* if and only if its application generates a valid (viz $\neq \perp$) model fragment:

r is applicable, iff $\forall m_0 \in \mathbb{M}_{r_0|_{mv_0}|_{mm}} : transform(m_0, r) \neq \perp$

❍

First we provide several definitions that are necessary for the subsequent steps to prove the applicability of a rule set.

Definition 4.1.2 (*dependsOn*(ov, r_i))
For an object variable *ov* at the right hand side of a rule r_i *dependsOn* yields to the set of object variables $\{ov_0, ..., ov_m\}$ of the left hand side from which the identifier resp. the key attributes of *ov* can be deterministically calculated. If $ov|_{oiv}$ has a constant value then *dependsOn*(ov, r_i) returns the empty set $\emptyset$.

$$dependsOn : (r|_{mv_1}|_{OVB}, \{r_i | r = r_0, ..., r_n, 0 \leq i \leq n\}) \rightarrow \mathscr{P}(r|_{mv_0}|_{OVB}) \cup \{\perp, \diamondsuit\}$$

$$dependsOn(ov, r_i) \mapsto \begin{cases} \emptyset & \text{if } ov|_{oiv} \text{ resp. all key attributes of ‚}ov\text{‘ are constant} \\ \{ov_0, ..., ov_m\} & \text{if } ov|_{oiv} \text{ resp. all key attributes of ‚}ov\text{‘ are one to one dependent on } ov_i|_{oiv} \text{ resp. on the key attributes of ‚}ov_i\text{‘} \\ \diamondsuit & \text{if } ov|_{oiv} \text{ resp. one of the key attributes of ‚}ov\text{‘ is equal to } \diamondsuit \\ \perp & \text{otherwise} \end{cases}$$

❍

Informally spoken *dependsOn* yields to the set of source object variables of a given rule of which the identity of a target object variable is one-to-one dependent.

Definition 4.1.3 (*attDependsOn*(ov, att, r_i))
Let be $r_i = (mv_0, mv_1)$, $ov \in mv_1|_{OVB}$, and $att \in ov|_{VV}|_a$.

$$attDependsOn(ov, att, r_i) = \begin{cases} \emptyset & \text{if } att = \text{const} \vee att = \diamondsuit \\ \{ov_0, \ldots, ov_n\} & \text{if } att \text{ can be computed from exactly the attribute terms of the object variables } \{ov_0, \ldots, ov_n\} \subseteq mv_0|_{OVB} \\ \perp & \text{otherwise} \end{cases}$$

❍

attDependsOn yields to the set of source object variables of a given rule on which an attribute value of a target object variable depends.

Definition 4.1.4 (*determines*: $(OV^1 \overset{mv_0}{\rightsquigarrow} OV^2)$)
Let mv_0 be the left hand side of a rule and $OV^1, OV^2 \subseteq mv_0|_{OVB}$. The relation $OV^1 \overset{mv_0}{\rightsquigarrow} OV^2$ holds if

$$\begin{aligned} &\forall m \in \mathbb{M}_{mv_0|_{mm}} : \\ &\quad \forall mfm \in MFM(m, mv_0) : \\ &\quad\quad \forall mfm_i, mfm_j \in mfm : \\ &\quad\quad\quad \left(\forall ov^1 \in OV^1 : mfm_i(ov^1) = mfm_j(ov^1) \Rightarrow \forall ov^2 \in OV^2 : mfm_i(ov^2) = mfm_j(ov^2)\right) \end{aligned}$$

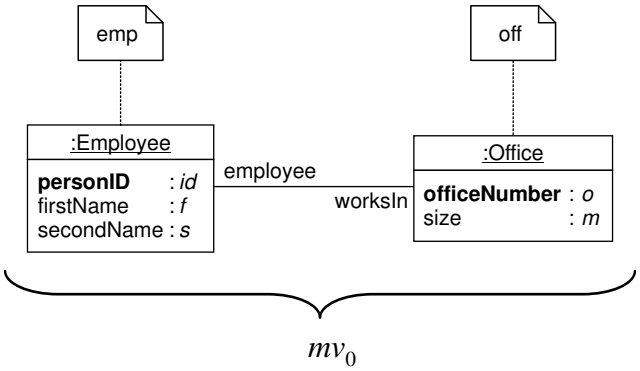

Figure 4.1: An example model variable

❍

Thus $OV^1 \overset{mv_0}{\rightsquigarrow} OV^2$ means: If the object variables in OV^1 do match to the same objects of the source model, then all object variables in OV^2 do also match to the respective same objects of the source model.

Lemma 4.1.1 ($\rightsquigarrow$ is reflexive)
Let mv_0 be a model variable and OV be a set of object variables according to Definition 4.1.4. Then it holds generally:

$OV \overset{mv_0}{\rightsquigarrow} OV$

I.e. $\rightsquigarrow$ is reflexive. ❑

Proof Sketch: Substitution into Definition 4.1.4 yields to a statement of the kind $A \Rightarrow A$ which is obviously true. ❑

Lemma 4.1.2 ($\rightsquigarrow$ is transitive)
Let mv_0 be a model variable and OV^1, OV^2, and OV^3 be sets of object variables according to Definition 4.1.4. Then it holds generally:

$OV^1 \overset{mv_0}{\rightsquigarrow} OV^2 \wedge OV^2 \overset{mv_0}{\rightsquigarrow} OV^3 \Rightarrow OV^1 \overset{mv_0}{\rightsquigarrow} OV^3$

I.e. $\rightsquigarrow$ is transitive. ❑

Proof Sketch: Substitution into Definition 4.1.4 yields to statements of the kind $A \Rightarrow B \wedge B \Rightarrow C$. Since then it holds $A \Rightarrow C$ Lemma 4.1.2 does hold, too. ❑

Theorem 4.1.1 (Verification technique for $\rightsquigarrow$ for two object variables)
Let mv_0 be an arbitrary but fix model variable and $OV^1, OV^2 \subseteq mv_0|_{OVB}$ as defined in Definition 4.1.4.

For $ov_1 \in OV^1$ and $ov_2 \in OV^2$ it does hold $ov_1 \overset{mv_0}{\rightsquigarrow} ov_2$, if it exists a sequence

$((ovae_0, \overline{ovae}_0), \ldots, (ovae_n, \overline{ovae}_n))$,with

$(i)\ \forall_{i\in\{0,\ldots,n\}} getova_{mv_0}(ovae_i, \overline{ovae}_i) \neq \perp$ (The *ovae* tuples are obj. variable assos,) $\wedge$
$(ii)\ ovae_0|_{ov} = ov_1$ (start of sequence is ov_1,) $\wedge$
$(iii)\ \overline{ovae}_n|_{ov} = ov_2$ (end of sequence is ov_2,) $\wedge$
$(iv)\ \forall_{i\in\{0,\ldots,n-1\}} : \overline{ovae}_i|_{ov} = ovae_{i+1}|_{ov}$ (sequence is coherent,) $\wedge$
$(v)\ \forall_{i\in\{0,\ldots,n\}} : \overline{ovae}_i|_{ae}|_m = (1,1)$ (sequence is path of to one associations.)

❑

Proof: Prove by induction over the length of the sequence l:

Induction Hypothesis: $l = 1$

Since the sequence has the length 1 $ovae_0|_{ov}$ and $\overline{ovae}_0|_{ov}$ are connected by exactly one object variable association according to (i), (ii), and (iii). (v) ensures that the corresponding class association has a multiplicity of $(1,1)$ from the class of $ovae_0|_{ov}$ to the class of $\overline{ovae}_0|_{ov}$. Thus $ovae_0|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_0|_{ov}$ holds for $l = 1$ (see also Definition 3.1.12 (3.14) on page 23).

Induction Step: $l + 1$

According to the induction hypothesis the proposition holds for sequences of the length l. Let $(ovae_0, \overline{ovae}_0), \ldots, (ovae_l, \overline{ovae}_l)$ be a sequence of the length $l+1$. From the induction hypothesis we assume that it holds $ovae_0|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_{l-1}|_{ov})$. According to an equivalent argumentation as in the induction hypothesis we can state that it holds $ovae_l|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_l|_{ov}$.

Considering (iv) we know that it holds: $\overline{ovae}_{l-1}|_{ov}) = ovae_l|_{ov}$

Together with Lemma 4.1.2 we can state that

$ovae_0|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_l|_{ov}$

❑

In Figure 4.1 a model variable with two object variables ov_{emp} and ov_{off} with the identifiers emp and off are shown. This model variable belongs to the metamodel mm_α of Figure 2.2 on page 7. In this example ov_{emp} determines ov_{off}, i.e. $\{ov_{emp}\} \overset{mv_0}{\rightsquigarrow} \{ov_{off}\}$, because according to the metamodel mm_α an employee works in exactly one office. Also some more trivial determines relations hold as for example: $\{ov_{emp}, ov_{off}\} \overset{mv_0}{\rightsquigarrow} \{ov_{off}\}$

Definition 4.1.5 (Applicability of a rule)
A rule r_i is *applicable* if and only if its application generates a valid (viz $\neq \perp$) model fragment:

r_i is applicable iff $\forall m_0 \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : apply(m_0, r_i, (\emptyset, \emptyset)) \neq \perp$

❍

Definition 4.1.6 (Applicability Equational System ($AES(ov_0, ov_1, k, l)$))
Let ov_0 and ov_1 be two object variables of the right side of a rule r_i with the same type *type*, i.e. $ov_0|_{otv} = ov_1|_{otv} = type$. Let $mfm^0 \in MFM(m, r_i|_{mv_0})$ be a model fragment match sequence for an arbitrary but fix model $m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$. Then $AES(ov_0, ov_1, k, l)$ denotes the following equational system:

$$\bigwedge_{\substack{att \in type|_A|_n / typ|_{Keys}|_n \\ \wedge mfm_k^1|_{match_o}(ov_0).att \neq \Diamond \\ \wedge mfm_l^1|_{match_o}(ov_1).att \neq \Diamond}} mfm_k^1|_{match_o}(ov_0).att = mfm_l^1|_{match_o}(ov_1).att \tag{4.1}$$

❍

Theorem 4.1.2 (Applicability of a rule)
A rule r_i is applicable, if all of the following three criteria do hold:

(i) the equational system *GL* according to Definition 3.4.5 is unambiguously resolvable.

(ii) $\forall ov \in r_i|_{mv_1}|_{OVB}$:

$$\begin{aligned}
&\big((dependsOn(ov, r_i) \notin \{\perp, \Diamond\} \\
&\quad \wedge \forall a \in ov|_{VV}|_a : dependsOn(ov, r_i) \overset{r_i|_{mv_0}}{\rightsquigarrow} attDependsOn(ov, a, r_i)) \\
&\vee dependsOn(ov, r_i) = \Diamond \\
&\vee \forall (a, t) \in ov|_{VV} : t = \Diamond\big)
\end{aligned}$$

(iii) For a single model fragment match mfm_i of a match sequence *mfm* we denote the solution of the equational defined in Definition 3.4.5 system with $GL'(ov_0, ov_1, i)$. Thereby $GL'(ov_0, ov_1, i)$ contains only the solutions for the identifiers and attributes of ov_0 and ov_1. $\forall ov_0, ov_1 \in r_i|_{mv_1}|_{OVB}$ with $ov_0|_{otv} = ov_1|_{otv}$: If

$$CES := (mfm_k^1|_{match_o}(ov_0)|_{oi} = mfm_l^1|_{match_o}(ov_1)|_{oi}) \wedge GL'(ov_0, ov_1, k) \wedge GL'(ov_0, ov_1, l)$$

is solvable, then it does hold:

$AES(ov_0, ov_1, k, l)$ does not increase the solution space of *CES*, i.e. $AES(ov_0, ov_1, k, l)$ doesn't restrict the solution of *CES*.

❑

(i) is a sufficient and necessary criterion to ensure that a model fragment transformation yields to a result different from $\perp$. The criteria (ii) and (iii) are sufficient but not necessary to ensure applicability.

Informally a rule r_i is applicable if the following three statements hold:

(i) All equations according to Definition 3.4.5 have a unique solution.

(ii) We can determine for every object variable $ov \in r_i|mv_1|OVB$ that at least one of the following statements hold:

- If the identity of *ov* is one-to-one depend to a set of object variables from the rule's left side we call this set *OV*. If the elements of *OV* are matched to the same objects of the left-hand model then every attribute of *ov* gets assigned the same value by the model fragment transformation.
- It holds for the identity of the object variable that $ov|_{oiv} = \diamondsuit$.
- All attribute values of *ov* are $\diamondsuit$.

(iii) If one newly generated object of the target model matches to two different object variables of the rules right model variable that are of the same type, and if the object identifier terms may lead to overlapping values, then those objects generated from those object variable have to be *mergeable*. Therefore (4.1) further determines that both matches yield to the same attribute values for this object, so that the merge operation does not fail. We demand that the constraints of *AES* do not further restrict the solution of *GL*. In this case we can be sure that there won't be any conflicting attribute values that result from these two object variables matching the same object.

Proof Sketch: Obviously there are two possibilities to obtain $\perp$ as the result of a rule application: either a model fragment transformation returns $\perp$ or a merge operation returns $\perp$ because different attribute values could not be merged. Thus postulation (i) ensures that no model fragment transformation within the rule application results in $\perp$. Further (ii) and (iii) ensure that no attribute values (different to $\diamondsuit$) conflict in the target model.

Subsidiary:

Let $mf = apply(m, r_i, (\emptyset, \emptyset))$
$= (\emptyset, \emptyset) \cup_{\mathfrak{m}} mft(mfm_0, r_i) \cup_{\mathfrak{m}} \cdots \cup_{\mathfrak{m}} mft(mfm_{|mfm|-1}, r_i)$

Obviously it holds that $mf = \perp$ if and only if (a) *mft* or (b) *merge* yields to $\perp$.

Case (a): This can only occur if *GL* is unresolvable according to Definition 3.4.5 (see Definition 3.4.6). This is excluded by (i).

Case (b): The merge of two model fragments mf_0, mf_1 yields to $\perp$ if it holds for these two model fragments: $\neg mergeable(mf_0|_{OB}, mf_1|_{OB})$. This can only occur if

- it holds for one $mf_i = \perp$. Though this is excluded by case (a). Or:

- two model fragments that originate from a model fragment transformation *mft* do contain an object with the same identifier but different attribute values. This is excluded by (ii) and (iii) (s. a. Definition 3.4.9).

❑

Lemma 4.1.3
Theorem 4.1.2 (iii) holds for one pair of object variables $ov_k, ov_j \in r_i|_{mv_1}|_{OVB}$ with $ov_k|otv = ov_j|otv$, if

$$\begin{aligned}
&\forall ov_k, ov_j \in r_i|_{mv_1}|_{OVB} \text{ with } k \neq j \wedge ov_k|_{otv} = ov_j|_{otv} \\
&\wedge (ov_k|_{oiv} \neq \diamondsuit \wedge \nexists (a,t) \in ov_k|_{VV} : a \in ov_k|_{otv}|_{Keys} \wedge t = \diamondsuit) \\
&\wedge (ov_j|_{oiv} \neq \diamondsuit \wedge \nexists (a,t) \in ov_j|_{VV} : a \in ov_j|_{otv}|_{Keys} \wedge t = \diamondsuit) : \\
&\quad \forall (a,t) \in ov_k|_{VV}, (a',t') \in ov_j|_{VV} \text{ with } a = a' \wedge a \notin ov_i|_{otv}|_{Keys} : t = t' \vee t = \diamondsuit \vee t' = \diamondsuit
\end{aligned}$$

❑

Proof: The proposition of Lemma 4.1.3 specifies exactly the case when the equational system *AES* from Definition 4.1.6 contains no or only trivial equations. I.e. at least one identificator term contains $\diamondsuit$, or the pairwise identical attributes are assigned to the same term or at least one attribute term is equal to $\diamondsuit$. ❑

From Lemma 4.1.3 it is obviously that for any two object variables in $r_i|_{mv_1}$ of the same type with an identity diverse to $\diamondsuit$ it has to hold that their pairwise identical attributes have either equal values or at least one of them has the value $\diamondsuit$. We don't have to consider any key attributes values, because whenever the key attributes differ it is ensured that they belong to different objects. This holds because Lemma 3.4.1 ensures that all objects that origin from one rule application are different.

We now give several examples to show how the applicability of a rule can be proved. First we take a look at our running example and proof the applicability of its rules. Theorem 4.1.2 (iii) is a case that is not relevant to our running example. Therefore we provide two additional examples where Theorem 4.1.2 (iii) has to be proven and Lemma 4.1.3 does not apply.

Example: Referring to our running example we now want two verify that the two rules r_0 and r_1 presented in Figure 2.5 (see page 9) are applicable. Therefore we have to prove for each of the two rules that the statements (i)-(iii) do hold.

Rule r_0

For better readability we denote the left hand employee object variable with ev_0, the left hand office variable with ov_0. Consequently the variables at the right side are called pv_1 (Person), cv_1 (Company), and rv_0 (Room).

Verification of statement (i) in Theorem 4.1.2:

$$
\begin{array}{rrll}
GL^0_{id}: (& match^0_o(ev_0)|oi & = \varepsilon & \text{skipped according to Def. 3.4.5)} \\
(& match^0_o(ov_0)|oi & = \varepsilon & \text{skipped according to Def. 3.4.5)} \\
GL^0_v: & match^0_o(ev_0).personId & = id & \\
 & match^0_o(ev_0).firstName & = f & \\
 & match^0_o(ev_0).secondName & = s & \\
 & match^0_o(ov_0).officeNumber & = o & \\
 & match^0_o(ov_0).size & = m & \\
GL^1_{id}: & pK(\{(personID, match^1_o(pv_1).personID)\}) & = match^1_o(pv_1)|oi & \\
 & \text{"ACME"} & = match^1_o(cv_1)|oi & \\
 & pK(\{(roomNo, match^1_o(rv_1).roomNo)\}) & = match^1_o(rv_1)|oi & \\
GL^1_v: & id & = match^1_o(pv_1).personID & \\
 & f \circ \text{" "} \circ s & = match^1_o(pv_1).fullName & \\
 & o & = match^1_o(pv_1).room & \\
 & o & = match^1_o(rv_1).roomNo & \\
 & m \cdot 10.76 & = match^1_o(rv_1).squareFt & \\
 & \diamondsuit & = match^1_o(rv_1).phone &
\end{array}
$$

We can resolve this equational system and get an unambiguous solution for the new objects' attributes and their identifiers:

$$
\begin{aligned}
match^1_o(pv_1)|oi &= pK(\{(personID, match^0_o(ev_0).id)\}) \\
match^1_o(cv_1)|oi &= \text{"ACME"} \\
match^1_o(rv_1)|oi &= pK(\{(roomNo, match^0_o(ov_0).officeNumber)\}) \\
match^1_o(pv_1).personID &= match^0_o(ev_0).personId \\
match^1_o(pv_1).fullName &= match^0_o(ev_0).firstName \circ \text{" "} \circ match^0_o(ev_0).secondName \\
match^1_o(pv_1).room &= match^0_o(ov_0).officeNumber \\
match^1_o(rv_1).roomNo &= match^0_o(ov_0).officeNumber \\
match^1_o(rv_1).squareFt &= match^0_o(ov_0).size \cdot 10.76 \\
match^1_o(rv_1).phone &= \diamondsuit
\end{aligned}
$$

Thus 4.1.5 (i) is satisfied.

Verification of statement (ii) in Theorem 4.1.2:

Obviously it does hold for pv_1:

$$\begin{aligned} dependsOn(pv_1, r_0) &= \{ev_0\} \\ attDependsOn(pv_1, personID, r_0) &= ev_0 \\ attDependsOn(pv_1, fullName, r_0) &= ev_0 \\ attDependsOn(pv_1, room, r_0) &= ov_0 \end{aligned}$$

Because of the to-one relation between Employee and Office in the metamodel in Figure 2.2 we know that it holds:

$\{ev_0\} \overset{r_0|mv_0}{\rightsquigarrow} \{ev_0, ov_0\}$

Thus it does hold:

$\forall a \in pv_1|VV|a : dependsOn(pv_1, r_0) \overset{r_0|mv_0}{\rightsquigarrow} attDependsOn(pv_1, a, r_0)$

Further it holds for cv_1:

$\forall (a,t) \in cv_0 : t = \diamondsuit$

It holds for ov_1:

$$\begin{aligned} dependsOn(ov_1, r0) &= \{ov_0\} \\ attDependsOn(ov_1, roomNo, r_0) &= ov_0 \\ attDependsOn(ov_1, squareFt, r_0) &= ov_0 \\ attDependsOn(ov_1, phone, r_0) &= \emptyset \end{aligned}$$

$\{ov_0\} \overset{r_0|mv_0}{\rightsquigarrow} \{ov_0\}$

Thus it does hold:

$\forall a \in ov_1|VV|a : dependsOn(ov_1, r_0) \overset{r_0|mv_0}{\rightsquigarrow} attDependsOn(ov_1, a, r_0)$

Thus (ii) does hold for the rule r_0.

Verification of statement (iii) in Theorem 4.1.2:

Since there are no two object variables of the same type in $r_0|mv_1$ (iii) does not apply and therefore r_0 is applicable.

Rule r_1

For better readability we denote the left hand office object variable with ov_0, the left hand phone variable with pv_0. The variable at the right side of r_1 is now called rv_1.

Verification of statement (i) in Theorem 4.1.2:

$$
\begin{array}{rcll}
GL^0_{id}: (& match^0_o(ov_0)|oi & = \varepsilon & \text{skipped according to Def. 3.4.5)} \\
(& match^0_o(pv_0)|oi & = \varepsilon & \text{skipped according to Def. 3.4.5)} \\
GL^0_v: & match^0_o(ov_0).officeNumber & = o & \\
 & match^0_o(ov_0).size & = \diamondsuit & \\
 & match^0_o(pv_0).phoneNumber & = pn & \\
GL^1_{id}: & pK(\{(roomNo, match^1_o(rv_1).roomNo)\}) & = match^1_o(rv_1)|oi & \\
GL^1_v: & o & = match^1_o(rv_1).roomNo & \\
 & \diamondsuit & = match^1_o(rv_1).squareFt & \\
 & \text{"+49 (89) 289 - "} \circ \text{ toStr}(pn) & = match^1_o(rv_1).phone &
\end{array}
$$

We can resolve this equational system and get an unambiguous solution for the new objects' attributes and their identifiers:

$$
\begin{aligned}
match^1_o(rv_1)|oi &= pK(\{(roomNo, match^0_o(ov_0).officeNumber)\}) \\
match^1_o(rv_1).roomNo &= match^0_o(ov_0).officeNumber \\
match^1_o(rv_1).squareFt &= \diamondsuit \\
match^1_o(rv_1).phone &= \text{"+49 (89) 289 - "} \circ \text{toStr}(match^0_o(pv_0).phoneNumber)
\end{aligned}
$$

Thus 4.1.5 (i) is satisfied.

Verification of statement (ii) in Theorem 4.1.2:

It does hold for rv_1:

$$
\begin{aligned}
dependsOn(rv_1, r_1) &= \{ov_0\} \\
attDependsOn(rv_1, roomNo, r_1) &= ov_0 \\
attDependsOn(rv_1, squareFt, r_1) &= \emptyset \\
attDependsOn(rv_1, phone, r_1) &= pv_0
\end{aligned}
$$

Because of the to-one relation between Phone and Office in the metamodel in Figure 2.2 we know that it holds:

$\{ov_0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \{pv_0, ov_0\}$

Thus it does hold:

$\forall a \in rv_1|VV|a : dependsOn(rv_1, r_1) \overset{r_1|_{mv_0}}{\rightsquigarrow} attDependsOn(rv_1, a, r_1)$

Thus (ii) does hold for the rule r_1.

Verification of statement (iii) in Theorem 4.1.2:

Since there are no two object variables of the same type in $r_1|mv_1$ (iii) does not apply and therefore r_1 is applicable.

❍

The following two examples are dedicated to a better understanding of Theorem 4.1.2 (iii). First we introduce a simple lemma we will need for the proof of postulation (iii).

Lemma 4.1.4
For $o_0, o_1 \in OB$ it does hold:

$\forall att \in o_0|_V|_a \cap o_1|_V|_a : o_0|_{oi} = o_0|_{oi} \Rightarrow o_0.att = o_1.att$

❑

Proof: The lemma obviously holds, because of Definition 3.1.10 (Object Allocation), which states that within an object allocation every object must have an unique identifier, and Definition 3.1.9 (Object). ❑

Example: In this example we show a case when Lemma 4.1.3 is not strong enough to prove that a given rule is applicable. Thereby we concentrate only on the prove of the statement demanded by Theorem 4.1.2 (iii), the statements (i) and (ii) are easy to prove and thus their proof is left to the reader.

Informally spoken statement (iii) deals with the case that the right side of a rule contains two or more object variables of the same type. Theorem 4.1.2 (ii) already determines that there may be no conflicts between generated objects that result from the same object variable. Lemma 4.1.3 determines that there are no conflicts between two object variables of the same type if one of them has a generated identity or the attribute values can't conflict because at least one of a pair is $\diamondsuit$.

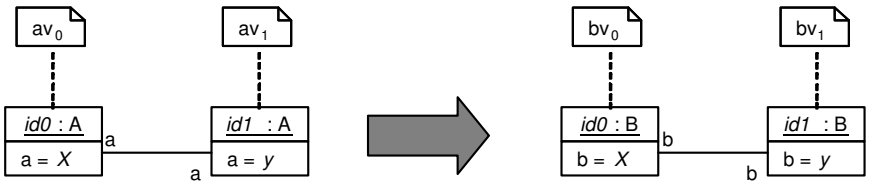

Figure 4.2: Rule r_{appl}

Let's consider rule r_{appl} that is depicted in Figure 4.2. As one can see Lemma 4.1.3 is not powerful enough to prove Theorem 4.1.2 (iii). Thus we have to prove this statement manually.

From Definition 3.4.5 we obtain the following set of equations, which are the equational system GL. Thereby, regarding an arbitrary but fix source model m we assume that mfm^0 is an arbitrary but fix model fragment sequence, with $mfm^0 \in MFM(m, r_{appl}|_{mv_0})$. GL holds for every h with $0 \leq h \leq |mfm^0| - 1$, i.e. it holds for every possible match in the source model.

$$
\begin{aligned}
GL^0_{id}: \; & mfm^0_h|_{match_o}(av_0)|_{oi} = id0 \\
& mfm^0_h|_{match_o}(av_1)|_{oi} = id1 \\
GL^0_v: \; & mfm^0_h|_{match_o}(av_0).a = x \\
& mfm^0_h|_{match_o}(av_1).a = y \\
GL^1_{id}: \; & id0 = mfm^1_h|_{match_o}(bv_0)|_{oi} \\
& id1 = mfm^1_h|_{match_o}(bv_1)|_{oi} \\
GL^1_v: \; & x = mfm^1_h|_{match_o}(bv_0).b \\
& y = mfm^1_h|_{match_o}(bv_1).b
\end{aligned}
$$

We can simplify this system into an equivalent system that provides the solutions for the attributes and identifiers of the target model objects. According to Definition 3.4.5 we regard only those that are solutions for identifiers or attributes for bv_0 or bv_1:

$$
\begin{aligned}
GL'(bv_0, bv_1, h): \; & mfm^1_h|_{match_o}(bv_0)|_{oi} = mfm^0_h|_{match_o}(av_0)|_{oi} \\
& mfm^1_h|_{match_o}(bv_1)|_{oi} = mfm^0_h|_{match_o}(av_1)|_{oi} \\
& mfm^1_h|_{match_o}(bv_0).b = mfm^0_h|_{match_o}(av_0).a \\
& mfm^1_h|_{match_o}(bv_1).b = mfm^0_h|_{match_o}(av_1).a
\end{aligned}
$$

We denote this system with $GL'(h)$ to indicate that it is the equational system for the model fragment match mfm_h^0 within the sequence mfm^0. Note that all these systems for mfm^0 are identical, except the sequence number h.

To prove Theorem 4.1.2 (iii) we first have to determine if the equational System *CES* is solvable:

$$
\begin{aligned}
& mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_l^1|_{match_o}(bv_1)|_{oi} \\
GL'(bv_0, bv_1, k): \; & mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_k^0|_{match_o}(av_0)|_{oi} \\
& mfm_k^1|_{match_o}(bv_1)|_{oi} = mfm_k^0|_{match_o}(av_1)|_{oi} \\
& mfm_k^1|_{match_o}(bv_0).b = mfm_k^0|_{match_o}(av_0).a \\
& mfm_k^1|_{match_o}(bv_1).b = mfm_k^0|_{match_o}(av_1).a \\
GL'(bv_0, bv_1, l): \; & mfm_l^1|_{match_o}(bv_0)|_{oi} = mfm_l^0|_{match_o}(av_0)|_{oi} \\
& mfm_l^1|_{match_o}(bv_1)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \\
& mfm_l^1|_{match_o}(bv_0).b = mfm_l^0|_{match_o}(av_0).a \\
& mfm_l^1|_{match_o}(bv_1).b = mfm_l^0|_{match_o}(av_1).a
\end{aligned}
$$

Please note that the variables of these equations are the identifiers and attributes of objects in the target model, i.e. these of the kind $mfm_{...}^1|_{match_o}....$

Regarding the equational system we can state that the system does hold, if

$$mfm_k^0|_{match_o}(av_0)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \tag{4.2}$$

This yields to true, if there are two matches where once av_0 matches an object of the source model and the at the other match av_1 matches to the same object. Obviously such two matches can (and in our symmetric case certainly do) exist.

Now let's consider *AES*. According to Definition 4.1.6 we obtain only a single equation for $AES(bv_0, bv_1, k, l)$:

$$AES: mfm_k^1|match_o(bv_0).b = mfm_l^1|match_o(bv_1).b$$

Regarding our equational system *CES* adding *AES* to it implies a new constraint:

$$mfm_k^0|_{match_o}(av_0).a = mfm_l^0|_{match_o}(av_1).a \tag{4.3}$$

Thus we have to prove that (4.3) does not further restrict the existing constraint postulated in (4.2). This does obviously hold, because of Lemma 4.1.4, which states that within an object allocation objects with the identical identifiers (see (4.2)) must have identical attribute values.

As one can see this means that the equation (4.2) already implies the equation (4.3). Therefore we can state that the rule r_{appl} is applicable.

❍

Example:

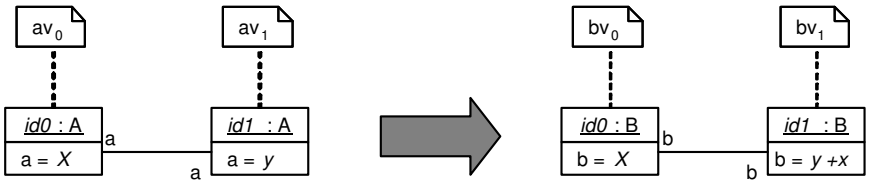

Figure 4.3: Rule r_{nappl}

Now let's consider a very similar rule depicted in Figure 4.3. Obviously the only difference is that now the attribute term in bv_1 has the value $y+x$ instead of y. Thus we can skip most of the considerations of the preceding example and directly present *CES*:

$$
\begin{aligned}
& & mfm_k^1|_{match_o}(bv_0)|_{oi} &= mfm_l^1|_{match_o}(bv_1)|_{oi} \\
GL'(bv_0,bv_1,k): & & mfm_k^1|_{match_o}(bv_0)|_{oi} &= mfm_k^0|_{match_o}(av_0)|_{oi} \\
& & mfm_k^1|_{match_o}(bv_1)|_{oi} &= mfm_k^0|_{match_o}(av_1)|_{oi} \\
& & mfm_k^1|_{match_o}(bv_0).b &= mfm_k^0|_{match_o}(av_0).a \\
& & mfm_k^1|_{match_o}(bv_1).b &= mfm_k^0|_{match_o}(av_0).a + mfm_k^0|_{match_o}(av_1).a \\
GL'(bv_0,bv_1,l): & & mfm_l^1|_{match_o}(bv_0)|_{oi} &= mfm_l^0|_{match_o}(av_0)|_{oi} \\
& & mfm_l^1|_{match_o}(bv_1)|_{oi} &= mfm_l^0|_{match_o}(av_1)|_{oi} \\
& & mfm_l^1|_{match_o}(bv_0).b &= mfm_l^0|_{match_o}(av_0).a \\
& & mfm_l^1|_{match_o}(bv_1).b &= mfm_l^0|_{match_o}(av_0).a + mfm_l^0|_{match_o}(av_1).a
\end{aligned}
$$

Regarding the equational system we can state again that the system does hold, if

$$mfm_k^0|_{match_o}(av_0)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \tag{4.4}$$

Again, according to Definition 4.1.6 we obtain only a single equation for $AES(bv_0,bv_1,k,l)$:

$$AES: mfm_k^1|match_o(bv_0).b = mfm_l^1|match_o(bv_1).b$$

Regarding our equational system *CES* adding *AES* to it implies a new constraint:

$$mfm_k^0|_{match_o}(av_0).a = mfm_l^0|_{match_o}(av_0).a + mfm_l^0|_{match_o}(av_1).a$$

According to (4.4) and Lemma 4.1.4 we know that

$$mfm_k^0|_{match_o}(av_0).a = mfm_l^0|_{match_o}(av_1).a$$

which yields to

$$mfm_l^0|_{match_o}(av_0).a = 0$$

I.e. the rule r_{nappl} is only applicable, if we can ensure that $mfm_l^0|_{match_o}(av_0).a = 0$. This does correspond with our intuition, because when we look at the rule we already can see that there might be no conflicting attribute values, if av_0 would always match to an object with an attribute value $a = 0$. However, this assumption does further constrain the number of possible solutions and therefore we can state that the rule r_{nappl} is not applicable.

❍

Theorem 4.1.3 (Applicability)
A sufficient but not necessary criterion for the applicability of a rule set r is the fulfillment of the following criterions for all rules r_i of the rule set r:

(i) r_i is applicable

(ii) $(\forall ov \in r_i|_{mv_1}|_{OVB}$ with $ov|_{oiv} \neq \diamondsuit$:
$\forall (a,t) \in ov|_{VV}$ with $a \notin ov|_{otv}|_{Keys} \wedge t \neq \diamondsuit$:
$\nexists j : j \neq i$
$\wedge \exists (a',t') \in ov'|_{VV} : t' \neq \diamondsuit$
$\wedge ov' \in r_j|_{mv_1}|_{OVB}$
$\wedge ov'|_{otv} = ov|_{otv} \wedge a' = a)$

❑

According to (i) every single rule of the rule set has to be applicable. Further according to (ii) for any two different rules there are no terms that potentially lead to mutual contradictory attribute values of one object.

Proof Sketch: *Subsidiary:*

Let $mf = transform(m,r)$
$= apply(m, r_{|r|-1}, apply(\cdots, apply(m, r_0, (\emptyset,\emptyset))\cdots))$

Obviously it holds that $mf = \perp$ if and only if (a) a rule application *apply* with empty source model fragment yields to $\perp$ during the rule set application or (b) the merge of a fragment, which has been newly created by a rule application, into the target fragment yields to $\perp$.

Case (a): Excluded by (i).

Case (b): This is excluded according to Definition 3.4.11, Definition 3.4.10 and Definition 3.4.9 by (ii).

❑

Example: With respect to our running example we can state that the rule set (r_0, r_1) depicted in Figure 2.5 (c.f. page 9) is applicable, because the two rules comply with the two postulations of Theorem 3.4.9. First all equations have a unique solution. Second there are no different object variables of the same type on the right side of any rule and all terms for attributes stem only from object variables which determine the identity of an object. Note that the term *o* within the object variable "Person" of rule r_0 is no problem as an Employee worksIn exactly one Office. Also the use of $\diamondsuit$ terms ensure that there are no contradictory terms for any two different rules. So the rule set is applicable. ❍

With these relative simple techniques one may prove that a given rule set is applicable. It is intentionally left to the developer of a rule set to ensure that the equations sets within each rule have a unique solution. Although it is possible to restrict the data types and their operations accordingly, we have the strong feeling that this will not result in a pragmatically usable solution.

4.2 Metamodel Conformance

The second important property of rules and rule sets is the property of metamodel conformance. Applicability ensures that a rule or rule set has a result diverse $\perp$. But this is not enough as the result is a model fragment that might not be conform to the target metamodel, i.e. it might not be a target model. In the following techniques that allow to prove metamodel conformance of a rule set are developed. Therefore especially the cardinalities and multiplicities have to be examined.

4.2.1 Basics

Definition 4.2.1 (Valid Model)
A model *m* is called valid if and only if it holds: $m \neq \perp$ ❍

Definition 4.2.2 (Model Transformation *transform*(m,r))
A *model transformation* is a rule set application

$$transform : \mathbb{M}_{mm_0} \times \mathbb{RW}_{mm_0,mm_1} \rightarrow \mathbb{MF}_{mm_1}$$
$$transform(m,r) \mapsto mf$$

hereby $consistent(mf, r|_{mv_1}|_{mm})$ must hold.

If necessary identifiers can be mapped consistently into the set of valid identifiers of the target model. ❍

Definition 4.2.3 (Metamodel Conform Rule Sets)
A rule set r is called *metamodel conform*, if its application is a model transformation for any arbitrary source model. I.e. the rule set generates only valid models with respect to the target metamodel. ❍

Lemma 4.2.1
The result of a the application of an applicable rule set $transform(m,r) \mapsto mf$ is no valid model, i.e. $mf \notin \mathbb{M}_{r_{0|_{mv_1}|_{mm}}}$, if and only if, association multiplicities in mf are violated (cf. (3.14)). ❑

Proof Sketch: According to Definition 3.1.12 there exist the following possible reasons for a model fragment mf to be not conform with the according metamodel mm:

(i) OB does not match to the metamodel mm

(ii) (3.14) is violated, i.e. cardinalities are higher or lower than association multiplicities do allow

(iii) (3.15) is violated, i.e. associations don't have an object from m at one end

(iv) OB is not $\diamondsuit$-free

(v) The result of *transform* yields to $\perp$.

Proposition (i) and (iii) do not hold which is ensured by the definition of model fragments and the construction of mf as a result of a rule set application. (iv) could be easily fixed by removing all $\diamondsuit$ values of attributes by some default values. (v) is excluded by the property that the rule set is applicable. ❑

Lemma 4.2.2
For every object o of a model fragment mf that has been generated by a rule application

$$apply(m, r_i, (\emptyset, \emptyset)) = mf$$

and every class association that is connected to the objects type (i.e. its class) it does hold:

$$\forall (o \in mf|_{OB}, ae \in AE \text{ with } o|_t \in AE|_c):$$
$$\exists \overline{OV} \in \mathscr{P}(r_i|_{mv_1}|_{OVB}) \setminus \{\emptyset\} \text{ with}$$
$$(i) \forall ov \in \overline{OV} : ov|_{otv} = o|_{ot} \wedge$$
$$(ii) \min_{ov \in \overline{OV}} \Big(\sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} \Big) \leq \sum_{\substack{oae:oae=(ae,o) \\ \wedge oae \in oa|_{OAE} \\ \wedge oa|_{OAT}=AE}} oa|_{card}$$

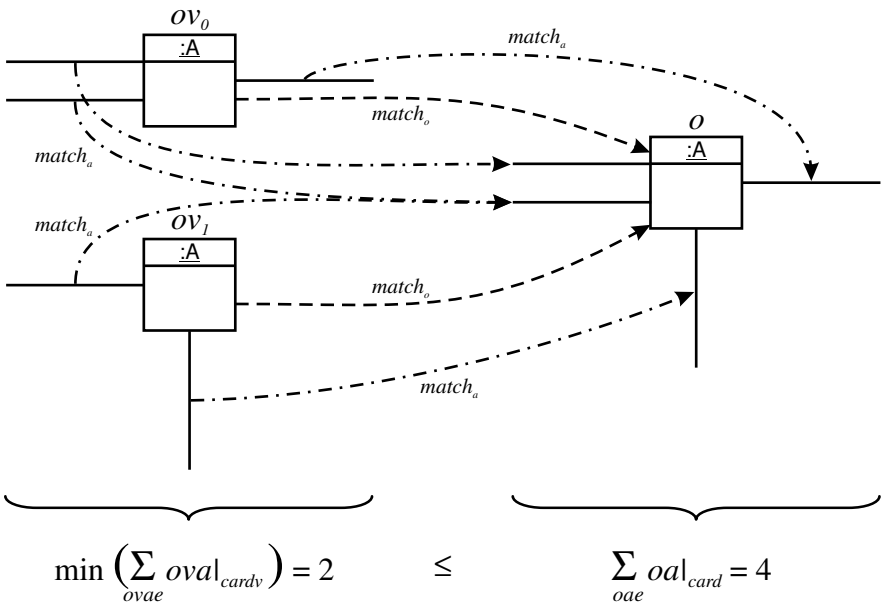

Figure 4.4: A sample clipping of an object stemming from two different object variables

❑

This means that the sum of the cardinalities of the outgoing associations from the same type of an object *o* is bigger or equal than the smallest sum of the cardinalities of the association variables that start in object variables *ov* from which *o* might originate.

In Figure 4.4 two object variables together with several outgoing object variable associations of the same type of a rule are shown on the left side. On the right side one object stemming from these two object variables is shown together with its object associations. Note that two object variable associations are merged in one object association in this example. The object itself is a result of the merge of two model fragments, too (cf. Theorem 3.4.1). Below the object and the object variable the two sums of Lemma 4.2.2 are shown.

Proof: *Subsidiary:* For a given rule the value of

$$\min_{ov \in \overline{OV}} \left(\sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} \right)$$

depends only on the type of the chosen object.

Reason: Because of (i) the proposition is evident directly from the definition.

Proof by induction over the generated model fragments $mf_{1,j}$

$$apply(m, r_i, (\emptyset,\emptyset)) = (\emptyset,\emptyset) \cup_{\mathfrak{m}} \bigcup_{\substack{\mathfrak{m} \\ j=0,\ldots,|mfm|-1}} mft(mfm_j, r_i)$$
$$= \bigcup_{\substack{\mathfrak{m} \\ j=0,\ldots,|mfm|-1}} mft(mfm_j, r_i)$$

Induction hypothesis, according Definition 3.4.10:

$mf_{1,0} = (\emptyset,\emptyset)$

The proposition holds obviously.

Induction step:

$$mf_{1,j} := (mf_{1,j-1} \cup_{\mathfrak{m}} \underbrace{mft(mfm_{j-1}, r_i)}_{=mf}) \tag{4.5}$$

According to the induction hypothesis the proposition holds for $mf_{1,j-1}$. Lemma 3.4.2 holds for all $o \in mf|_{OB}$, because every model fragment transformation *mft* is also a model fragment relation.

Let $(o \in mf_{1,j}, AE : o|_{ot} \in AE|_c, ae \in AE)$ be arbitrary. Because of (4.5) and Lemma 3.4.4 (ii) it holds that:

$o \in mf_{1,j-1}|_{OB} \cup mf|_{OB}$

Case 1: $o \in mf_{1,j-1}|_{OB}$ Statement (i) holds trivially.

$$\sum_{\substack{oae:oae=(ae,o) \\ \wedge oae \in oa|_{OAE} \\ \wedge oa|_{OAT}=AE \\ \text{for } o \in mf_{1,j}}} oa|_{card} \geq \sum_{\substack{oae:oae=(ae,o) \\ \wedge oae \in oa|_{OAE} \\ \wedge oa|_{OAT}=AE \\ \text{für } o \in mf_{1,j-1}}} oa|_{card} \qquad \text{(Lemma 3.4.4 } (i)\text{)}$$

$$\geq \underbrace{\min_{ov \in \overline{OV}} \Big(\sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} \Big)}_{=const.} \qquad \begin{array}{l}\text{(Assumption and} \\ \text{subsidiary from above)}\end{array}$$

Case 2 $o \in mf|_{OB}$ Because of Lemma 3.4.2 it holds:

$\exists ov \in r_i|_{mv_1}|_{OVB}$ with

(a) $o|_{ot} = ov|_{otv}$

$$(b)\quad \sum_{\substack{oae:oae=(ae,o)\\ \wedge oae\in oa|_{OAE}\\ \wedge oa|_{OAT}=AE\\ \text{for } o\in mf_{1,j-1}}} oa|_{card} = \sum_{\substack{ovae:ovae=(ae,ov)\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv}$$

$$\geq \min_{ov\in\overline{OV}}\Big(\sum_{\substack{ovae:ovae=(ae,ov)\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv}\Big)$$

$ov \in \overline{OV}$, since

$$\left|\overline{OV}\right| \overset{Def.}{\geq} 1 \Rightarrow \exists ov' \in \overline{OV}$$

$$\text{with } \min_{ov\in\overline{OV}}\Big(\sum_{\substack{ovae:ovae=(ae,ov)\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv}\Big) = \sum_{\substack{ovae:ovae=(ae,ov')\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv}$$

Case 2.1 $ov = ov'$ The proposition is evident directly.

Case 2.2 $ov \neq ov'$

$$\sum_{\substack{ovae:ovae=(ae,ov)\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} = \sum_{\substack{ovae:ovae=(ae,ov')\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv}$$

Because of (a) it holds: $ov \in \overline{OV}$

$$\overset{\text{Lemma 3.4.4 } (i)}{\Rightarrow} \sum_{\substack{oae:oae=(ae,o)\\ \wedge oae\in oa|_{OAE}\\ \wedge oa|_{OAT}=AE\\ \text{for } o\in(mf_{1,j-1}\cup_{\mathfrak{m}} mf)|_{OB}}} oa|_{card} \geq \sum_{\substack{oae:oae=(ae,o)\\ \wedge oae\in oa|_{OAE}\\ \wedge oa|_{OAT}=AE\\ \text{for } o\in mf|_{OB}}} oa|_{card}$$

$$\Rightarrow \quad \forall(o, AE : o|_{ot} \in AE|_c, ae \in AE)$$
$$\exists\overline{OV} \in \mathscr{P}(r_i|_{mv_1}|_{OVB}) \setminus \{\emptyset\}$$
$$\text{with } \overline{OV} = \Big\{ov : ov|_{otv} = o|_{ot}$$
$$\wedge \sum_{\substack{oae:oae=(ae,o)\\ \wedge oae\in oa|_{OAE}\\ \wedge oa|_{OAT}=AE}} oa|_{card} \geq \min_{ov\in\overline{OV}}\Big(\sum_{\substack{ovae:ovae=(ae,ov)\\ \wedge ovae\in ova|_{OVAE}\\ \wedge ova|_{OVAT}=AE}} ova|_{cardv}\Big)\Big\}$$

Note: Case 1 and case 2 do overlap, which is of no relevance, though. ❏

Definition 4.2.4 (*ubConform*(mf, mm))
For a metamodel *mm* and an appropriate model fragment *mf* *ubConform*(mf, mm) is a predicate so that it holds

$$
\begin{aligned}
&ubConform(mf, mm) \\
&\quad \Leftrightarrow \\
&mf \neq \bot \wedge \forall AE \in mm|_{CA} : \forall ae \in AE : (lb, ub) = ae|_m : \\
&\qquad \forall o \in mf|_{OB} \text{ with } o|_{ot} = ae|_c : \sum_{oa:(oppositeEnd(AE,ae),o)\in oa|_{OAE}} oa|_{card} \leq ub
\end{aligned}
$$

❍

ubConform(mf, mm) holds, if and only if, the sum of all cardinalities of outgoing object associations for every object in a model fragment *mf* are equal or below the upper bounds of the regarding multiplicities in the metamodel *mm*.

Definition 4.2.5 (*lbConform*(mf, mm))
For a metamodel *mm* and an appropriate model fragment *mf* *lbConform*(mf, mm) is a predicate so that it holds

$$
\begin{aligned}
&lbConform(mf, mm) \\
&\quad \Leftrightarrow \\
&mf \neq \bot \wedge \forall AE \in mm|_{CA} : \forall ae \in AE : (lb, ub) = oppositeEnd(AE, ae)|_m : \\
&\qquad \forall o \in mf|_{OB} \text{ with } o|_{ot} = ae|_c : lb \leq \sum_{\substack{oa\in mf|_{OA} \text{ with} \\ oa|_{oat}=AE \\ \wedge(ae,o)\in oa|_{OAE}}} oa|_{card}
\end{aligned}
$$

❍

lbConform(mf, mm) holds if and only if the sum of all cardinalities of out-going object associations for every object in a model fragment *mf* are equal or above the lower bounds of the regarding multiplicities in the metamodel *mm*.

Definition 4.2.6 (*MFM*$^1(m_0, r_i)$)
Let *MFM*$^1(m_0, r_i)$ be the set of model fragment matches mfm_μ^1 that occur during the application of *apply*$(m_0, r_i, (\emptyset, \emptyset))$, according to Definition 3.4.5, whereby $m_0 \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$. ❍

Note: *MFM*^{1} is deterministic, because any sequence of model fragment matches consists of the same elements according to the definition of *apply* (see also Theorem 3.4.8).

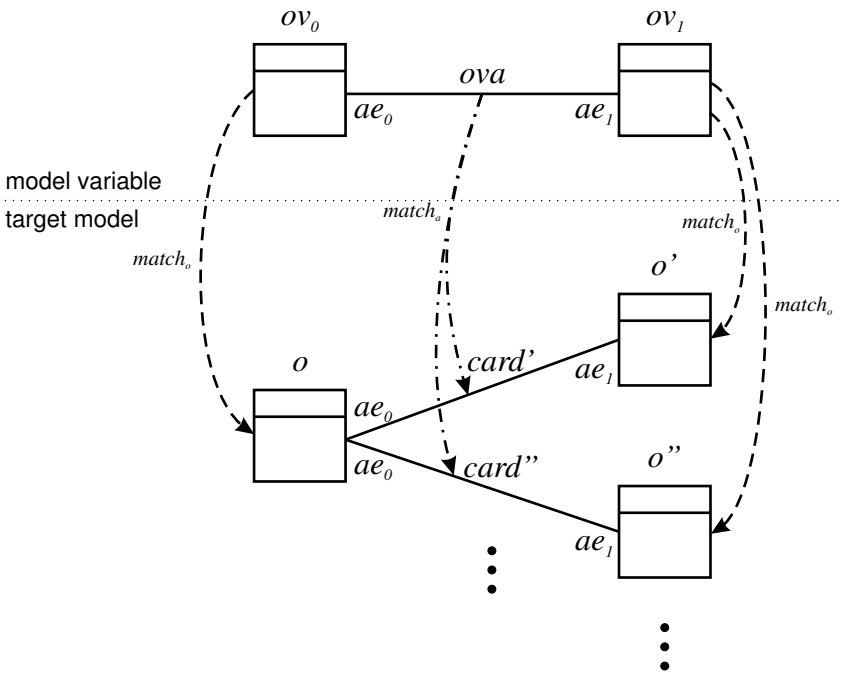

Figure 4.5: Object variable associations and their results

Definition 4.2.7 (*numAssos*(r_i, m_0, o, ova, ae_1))
Let r_i be a rule and $m_0 \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$. Further $ova \in r_i|_{mv_1}|_{OVA}$ and $ae_1 \in ova|_{OVAE}|_{ae}$. $o \in apply(m_0, r_i, (\emptyset, \emptyset))$, whereby $MFM^1(m_0, r_i)$ denotes the set of model fragment matches according to Definition 4.2.6.

$$
\begin{aligned}
&numAssos(r_i, m_0, o, ova, ae_1) \\
&\quad = ova|_{cardv} \cdot \big|\{oa : \exists mfm^1_\mu \in MFM^1(m_0, r_i) \\
&\qquad\qquad \text{with } mfm^1_\mu|_{match_a}(ova) = oa \\
&\qquad\qquad \wedge (oppositeEnd(ova|_{OVAT}, ae_1), o) \in oa|_{OAE} \\
&\qquad\qquad \wedge ae_1 \in ova|_{OVAT}|_{ae}\}\big|
\end{aligned}
$$

❍

For an object *o* in a generated target model and an association variable *ova* at the right hand side of a rule *numAssos* yields to the number of all outgoing associations from *o* that are of the same type like *ova* and that originate directly from *ova*. That means for the respective associations and the rule there exists a model fragment match mfm^1_μ containing a bijective mapping $mfm^1_\mu|_{match_a}$, that transform *ova* bijectively into *oa*. Figure 4.5 shows an example with an object variable association and two object associations generated from it.

Definition 4.2.8 ($getova_{mv}(ovae_0, ovae_1)$)
For a given model variable $mv = (mm, OVB, OVA)$ we define

$$getova_{mv} : OVA|_{OVAE} \times OVA|_{OVAE} \to OVA \cup \{\bot\}$$
$$getova_{mv}(ovae_0, ovae_1) \mapsto \begin{cases} ova & \text{if } \exists ova \text{ with } \{ovae_0, ovae_1\} = ova|_{OVAE} \\ \bot & \text{otherwise} \end{cases}$$

❍

getova yields to the object variable association that contains the given object variable association ends. If no object variable association with these end(s) exists $\bot$ is returned.

4.2.2 Verification techniques for upper bounds conformance

Definition 4.2.9 (Upper bounds conform rule set ($ubConform(r)$))
A rule set r with a source meta model $mm_0 = r_0|_{mv_0}|_{mm}$ and target meta model $mm_1 = r_0|_{mv_1}|_{mm}$ is called *upper bounds conform,* if it holds:

$$\forall m \in \mathbb{M}_{mm_0} : ubConform(transform(m, r), mm_1)$$

For short we write: $ubConform(r)$. ❍

Definition 4.2.10 (Similarity of identifier terms $ov_0|_{oiv} \sim ov_1|_{oiv}$)
For two identifier terms the predicate $\sim$ which is used with infix notation holds iff:

$$oiv_0 \sim oiv_1 :\Leftrightarrow (oiv_0 \neq \Diamond \wedge oiv_1 \neq \Diamond \wedge oiv_0, oiv_1 \text{ are } n\text{-tuples})$$

❍

Definition 4.2.11 ($maxRedundant(ova, r_i, r)$)
Let $ova \in r_i|_{mv_1}|_{OVA|_{OVB}}$ and $r_i \in r$. $maxRedundant(ova, r_i, r)$ is a predicate that holds, iff

$$\forall m \in \mathbb{M}_{r_i|_{mv_0|mm}} :$$
$$\{oa : (\exists mfm^1_\mu \in MFM^1(m, r_i) : mfm^1_\mu|_{match_a}(ova) = oa) \wedge (oa \in transform(m, r)|_{OA})\}$$
$$\subseteq$$
$$\{oa : oa|_{OAT} = ova|_{OVAT} \wedge oa \in transform(m, r \setminus \{r_i\})|_{OA}\}$$

❍

$maxRedundant(ova, r_i, r)$ denotes that the object variable association *ova* in r_i does not produce any object associations in the target model that wouldn't be produced by the other rules of r.

Definition 4.2.12 (*maxCard*(AE, ae_1, r))

$maxCard(AE, ae_1, r) \mapsto n \in \mathbb{N}_0^\infty$ with $ae_1 \in AE$

$$maxCard(AE, ae_1, r) = \max_{m \in \mathbb{M}_{r_0|_{mv_0}|_{mm}}} \Big(\max_{o \in transform(m,r)|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae_1),o) \in oa|_{OAE}} oa|_{card} \Big) \Big)$$

❍

$maxCard(AE, ae_1, r)$ yields to the maximum number of associations of the type AE that can go out of any object that can be created by any application of r. Thereby the end ae_1 specifies the opposite end of this association seen from o.

Theorem 4.2.1
Let r be a rule set and $mm = r_0|_{mv_1}|_{mm}$. Then it holds:

$$\forall AE, ae \text{ with } AE \in mm|_{CA}, ae \in AE : maxCard(AE, ae, r) \leq ub, \text{ with } ae|_m = (lb, ub)$$
$$\Rightarrow$$
$$ubConform(r)$$

❑

Proof:

$$\begin{aligned}
& \forall AE, ae \text{ with } AE \in mm|_{CA}, ae \in AE : maxCard(AE, ae, r) \leq ub, \\
& \text{with } ae|_m = (lb, ub) \\
\overset{\text{Def. 4.2.12}}{\Rightarrow} \quad & \max_{m \in \mathbb{M}_{r_0|_{mv_0}|_{mm}}} \Big(\max_{o \in transform(m,r)|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae),o) \in oa|_{OAE}} oa|_{card} \Big) \Big) \leq ub \\
& \text{with } ae|_m = (lb, ub) \\
\overset{\text{"}\max_i \max_j \sum_{k(i,j)} card \geq \forall_i \forall_j \sum_{k(i,j)} card\text{"}}{\Rightarrow} \quad & \forall AE \in mm|_{CA} : \forall ae \in AE : \forall m \in \mathbb{M}_{r_0|_{mv_0}|_{mm}} : \\
& \forall o \in transform(m,r)|_{OB} \text{ with } o|_{ot} = ae|_c : \\
& \sum_{oa:(oppositeEnd(AE,ae),o) \in oa|_{OAE}} oa|_{card} \leq ub \text{ with } ae|_m = (lb, ub) \\
\overset{\text{Def. 4.2.4}}{\Rightarrow} \quad & \forall m \in \mathbb{M}_{r_0|_{mv_0}|_{mm}} : ubConform(transform(m,r), mm) \\
\overset{\text{Def. 4.2.9}}{\Rightarrow} \quad & ubConform(r)
\end{aligned}$$

❑

Thus $maxCard(AE, ae, r)$ can be used to prove that a given rule set is *ubConform*. However *maxCard* usually cannot be directly computed. Therefore we have to provide appropriate estimations for *maxCard*.

Definition 4.2.13 ($maxVarCard(ova, ae_1, r_i)$**)**

$$maxVarCard(ova, ae_1, r_i) = \max_{m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}} \left(\max_{o \in apply(m, r_i, (\emptyset,\emptyset))|_{OB}} (numAssos(r_i, m, o, ova, ae_1)) \right)$$

❍

$maxVarCard$ yields to the maximum possible number of associations starting at an object that have been created from the same object variable association ova. This holds for an arbitrary source model.

Definition 4.2.14 (Model Variable is substructure of a Model Variable ($mv_{sub} \sqsubseteq mv$**))**
Given two model variables mv and mv_{sub} we call mv_{sub} a *substructure* of mv (for short: $mv_{sub} \sqsubseteq mv$) if it holds:

$$\begin{aligned}
&\exists corresponds : mv_{sub}|_{OVB} \to mv|_{OVB} \text{ with } corresponds \text{ is injective} \quad \wedge \\
&\forall ova_i \in mv_{sub}|_{OVA} : \\
&\quad \exists_1 ova_j \in mv|_{OVA} \text{ with} \\
&\qquad ova_j|_{OVAT} = ova_i|_{OVAT} \wedge \\
&\qquad ova_j|_{cardv} = ova_i|_{cardv} \wedge \\
&\qquad ova_j|_{OVAE}|_{ae} = ova_i|_{OVAE}|_{ae} \wedge \\
&\qquad \forall ovae' \in ova_j|_{OVAE}, ovae'' \in ova_i|_{OVAE} : \\
&\qquad\quad ovae'|_{ae} = ovae''|_{ae} \Rightarrow ovae'|_{ov} = corresponds(ovae''|_{ov})
\end{aligned}$$

Two model variables mv_a and mv_b are *congruent*, if:

$$mv_a \cong mv_b :\Leftrightarrow mv_a \sqsubseteq mv_b \wedge mv_b \sqsubseteq mv_a$$

Let mv_a and mv_b be model variables:

$$mv_a \sqsubset mv_b :\Leftrightarrow mv_a \sqsubseteq mv_b \wedge \neg(mv_a \cong mv_b)$$

❍

Thus $mv_{sub} \sqsubseteq mv$ holds if mv_{sub} is a part of the model variable mv. Thereby the terms in the attributes may differ. In Figure 4.13 (see page 90) the left model variable of r_1 is a substructure of the left model variable of r_0, i.e. we can state $r_1|_{mv_0} \sqsubseteq r_0|_{mv_0}$.

Definition 4.2.15 ($maxRelevant(r, r_i, AE, ae_1, ova', ov_0, ov'_0)$**)**

$$
\begin{aligned}
maxRelevant(r, r_i, AE, ae_1, ova', ov_0, ov'_0) :\Leftrightarrow\ & r_i \in r \\
& \wedge\, ova' \in r_i|_{mv_1}|_{OVA} \\
& \wedge\, ova'|_{OVAE}|_{ae} = AE \\
& \wedge\, (oppositeEnd(AE, ae_1), ov'_0) \in ova'|_{OVAE} \\
& \wedge\, ov'_0|_{otv} = ov_0|_{otv} \\
& \wedge\, ov'_0|_{oiv} \sim ov_0|_{oiv} \\
& \wedge\, \neg\big(maxRedundant(ova', r_i, r) \\
& \qquad \wedge\, \nexists r_j : (\, j < i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \\
& \qquad\qquad \wedge\, maxRedundant(ova', r_i, (r_j, r_i)))\big)
\end{aligned}
$$

❍

The predicate *maxRelevant* decides whether an outgoing object variable association is relevant in the context of a given rule set r. This holds, if we may not neglect this association without changing the possible results of a model transformation.

maxRelevant gets a rule set r and a rule r_i as a context. Further the type of the association we consider is specified by an class association AE, a class association end ae_1 and an object variable ov_0. This triple specifies an outgoing object variable association starting at ov_0 with the "opposite" end ae_1.

maxRelevant compares this specification for similar object variable associations. If the following is all true, then the outgoing object variable association specified by ova' and ov'_0 is relevant:

(*i*) ova' and ov'_0 are part of the right hand side of the rule r_i.

(*ii*) The object variable association has the according type AE.

(*iii*) The object variable ov'_0 is connected to ova' at the opposite side of ae_1.

(*iv*) The type of ov'_0 is the same as the type of ov_0 and both object variables are similar (i.e. have similar identifier terms).

(*v*) We do not consider the given object variable association, if it is maxRedundant and there exists no rule r_j with $j < i$ which has a congruent left hand model variable and to which r_i is maxRedundant. In this special case the association variables in the two rules would be mutual redundant and we make the policy to count only those in the rule with the smaller index.

Lemma 4.2.3
Let r be a rule set and AE a class association with $AE \in r_0|_{mv_1}|_{mm}|_{CA}$ and ae_1 a class association end $ae_1 \in AE$.

$$\begin{aligned}
&\forall r_k, ov_0, ae_1, AE \text{ with } r_k \in r, \\
&\qquad AE \in CA_1, \\
&\qquad ae_1 \in AE, \\
&\qquad (oppositeEnd(AE, ae_1), ov_0) \in r|_{mv_1}|_{OVA}|_{OVAE} \\
&maxCard(AE, ae_1, r) \leq \begin{cases}
\sum\limits_{\substack{ova' : ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae} = AE \\ \wedge (oppositeEnd(AE,ae_1), ov_0) \in ova'|_{OVAE}}} maxVarCard(ova', ae_1, r_k) & \text{if } ov_0|_{oiv} = \diamondsuit \\
\sum\limits_{\substack{r_i, ova', ov_0' : \\ maxRelevant(r, r_i, AE, ae_1, ova', ov_0, ov_0')}} maxVarCard(ova', ae_1, r_i) & \text{otherwise}
\end{cases}
\end{aligned}$$

❑

Proof: *Case I:* $ov_0|_{oiv} = \diamondsuit$

$$\begin{aligned}
&\sum_{\substack{ova' : ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae} = AE \\ \wedge (oppositeEnd(AE,ae_1), ov_0) \in ova'|_{OVAE}}} maxVarCard(ova', ae_1, r_k) \\
= &\sum_{\substack{ova' : ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae} = AE \\ \wedge (oppositeEnd(AE,ae_1), ov_0) \in ova'|_{OVAE}}} \max_{m \in \mathbb{M}_{r_k|_{mv_0}|_{mm}}} \\
&\qquad \Big(\max_{o \in apply(m, r_k, (\emptyset,\emptyset))|_{OB}} (numAssos(r_k, m, o, ova', ae_1)) \Big) \\
= &\sum_{\substack{ova' : ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae} = AE \\ \wedge (oppositeEnd(AE,ae_1), ov_0) \in ova'|_{OVAE}}} \max_{m \in \mathbb{M}_{r_k|_{mv_0}|_{mm}}} \Big(\max_{o \in apply(m, r_k, (\emptyset,\emptyset))|_{OB}} (ova'|_{cardv} \cdot \\
&\qquad \Big| \{ oa : \exists mfm_\mu^1 \in MFM^1(m, r_k) \\
&\qquad\qquad \text{with } mfm_\mu^1|_{match_a}(ova') = oa \\
&\qquad\qquad \wedge (oppositeEnd(ova'|_{OVAT}, ae_1), o) \in oa|_{OAE} \\
&\qquad\qquad \wedge ae_1 \in ova'|_{OVAT}|_{ae} \} \Big| \Big))
\end{aligned}$$

$$
\begin{aligned}
& \overset{ov_0|_{oiv}=\lozenge \Rightarrow \exists_1 mfm_\mu^1}{=} && \sum_{\substack{ova':ova'\in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae}=AE \\ \wedge(oppositeEnd(AE,ae_1),ov_0)\in ova'|_{OVAE}}} ova'|_{cardv} \\
& \overset{\text{Def. 3.4.10}}{\geq} && \\
& && \max_{m\in\mathbb{M}_{r_0|_{mv_0}|_{mm}}} \Big(\max_{o\in apply(m,r_k,(\emptyset,\emptyset))|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card} \Big) \Big) \\
& \overset{ov_0|_{oiv}=\lozenge}{=} && \max_{m\in\mathbb{M}_{r_0|_{mv_0}|_{mm}}} \Big(\max_{o\in\bigcup_{\mathfrak{m}\, r_i\in r} apply(m,r_i,(\emptyset,\emptyset))|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card} \Big) \Big) \\
& \overset{Thm.3.4.10}{=} && \max_{m\in\mathbb{M}_{r_0|_{mv_0}|_{mm}}} \Big(\max_{o\in transform(m,r)|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card} \Big) \Big) \\
& = && maxCard(AE,ae_1,r)
\end{aligned}
$$

Case II: $ov_0|_{oiv} \neq \lozenge$

Subsidiary 1: Generally it holds that:

$$\max_k \Big(\sum_l f(k,l) \Big) \leq \sum_l \max_k f(k,l) \tag{4.6}$$

Subsidiary 2: Let be $m \in \mathbb{M}_{r|_{r_0}|_{mv_0}|_{mm}}$. Let $o \in m$ be an arbitrary object. Let ov_0 be an arbitrary object variable which may have generated o.

$$
\begin{aligned}
& \big\{ oa : oa \in \bigcup_{\substack{\mathfrak{m} \\ r_i:r_i\in r}} apply(m,r_i,(\emptyset,\emptyset)) \\
& \quad \wedge (oppositeEnd(AE,ae_1),o) \in oa|_{OAE} \big\} \\
& \subseteq \\
& \big\{ oa : (\exists r_i \in r : \exists mfm_\mu^1(ova) = oa \\
& \quad \wedge (oppositeEnd(ova|_{OVAT},ae_1),o) \in o) \in oa|_{OAE} \\
& \quad \wedge ae_1 \in ova|_{OVAT}|_{ae_1} \\
& \quad \wedge ova \in \{ova' : \exists ov_0' \in r_i|_{OVB} \wedge maxRelevant(r,r_i,AE,ae_1,ova',ov_0,ov_0')\} \big\}
\end{aligned}
\tag{4.7}
$$

Equation (4.7) holds because every object association contained in the first set is part of the second set. This is the direct consequence of the defintion of *maxRedundant* (Def. 4.2.11) and the definition of *merge* (Def. 3.4.9). The intention of *maxRedundant* is to identify redundant object variable associations. These identified object variable associations are exactly those which are removed by OA_2 within the *merge* by definition.

$$
\begin{aligned}
& \sum_{\substack{r_i,ova',ov_0': \\ maxRelevant(r,r_i,AE,ae_1,ova',ov_0,ov_0')}} maxVarCard(ova',ae_1,r_i) \\
= & \sum_{\substack{r_i,ova',ov_0': \\ maxRelevant(r,r_i, \\ AE,ae_1,ova',ov_0,ov_0')}} \max_{m\in\mathbb{M}_{r_i|mv_0|mm}} \Big(\max_{o\in apply(m,r_i,(\emptyset,\emptyset))|_{OB}} (numAssos(r_i,m,o,ova',ae_1)) \Big) \\
= & \sum_{r_i:r_i\in r} \sum_{\substack{ova',ov_0': \\ maxRelevant(r,r_i, \\ AE,ae_1,ova',ov_0,ov_0')}} \max_{m\in\mathbb{M}_{r_i|mv_0|mm}} \Big(\max_{o\in apply(m,r_i,(\emptyset,\emptyset))|_{OB}} (numAssos(r_i,m,o,ova',ae_1)) \Big) \\
= & \sum_{r_i:r_i\in r} \sum_{\substack{ova',ov_0': \\ maxRelevant(r,r_i, \\ AE,ae_1,ova',ov_0,ov_0')}} \max_{m\in\mathbb{M}_{r_i|mv_0|mm}} \Big(\max_{o\in apply(m,r_i,(\emptyset,\emptyset))|_{OB}} (ova'|_{cardv} \cdot \\
& \Big| \{ oa : \exists mfm^1_\mu \in MFM^1(m,r_i) \\
& \qquad \text{with } mfm^1_\mu|_{match_a}(ova') = oa \\
& \qquad \wedge (oppositeEnd(ova'|_{OVAT},ae_1),o) \in oa|_{OAE} \\
& \qquad \wedge ae_1 \in ova'|_{OVAT}|_{ae} \} \Big|)) \\
\overset{(4.6),(4.7)}{\geq} & \max_{m\in\mathbb{M}_{r_0|mv_0|mm}} \Big(\max_{o\in\bigcup_{\substack{m \\ r_i\in r}} apply(m,r_i,(\emptyset,\emptyset))|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card} \Big) \Big) \\
\overset{\text{Thm. 3.4.10}}{=} & \max_{m\in\mathbb{M}_{r_0|mv_0|mm}} \Big(\max_{o\in transform(m,r)|_{OB}} \Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card} \Big) \Big) \\
= & \; maxCard(AE,ae_1,r)
\end{aligned}
$$

❑

Lemma 4.2.3 states that the the maximum number of associations of the type *AE* that can go out of any object that can be created by any application of r ($maxCard(AE,ae,r)$) is less or equal than the sum over all *maxVarCard*s of all non-redundant object association variables.

Definition 4.2.16 ($maxmatch(mm,mv,\{ov_i,...,ov_j\})$**)**

$maxmatch(mm,mv,\{ov_i,...,ov_j\}) \mapsto \mathbb{N}_0^\infty$ with $\{ov_i,...,ov_j\} \subseteq mv|_{OVB}$

For a metamodel *mm*, a model variable *mv*, and a set of object variables that are in *mv* and marked as "fix" *maxmatch* yields to the maximum number of possible matches of the model variable in an arbitrary model, where the fix object variables are regarded as arbitrary but fix. ❍

Note: For the following special cases it holds:

$maxmatch(mm,mv,\perp) = 0$

$maxmatch(mm,mv,\emptyset) = 0$

Example: In the following we explain *maxmatch* with a simple example. Figure 4.6 shows an example metamodel with three classes A, B, and C. Not relevant details like attribute definitions, role names, or terms for identificators are not shown in the example.

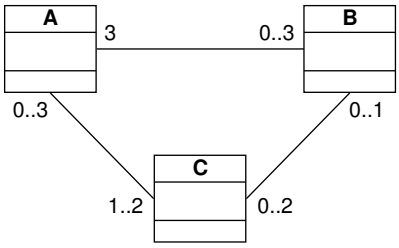

Figure 4.6: An example metamodel *mm* for *maxmatch*

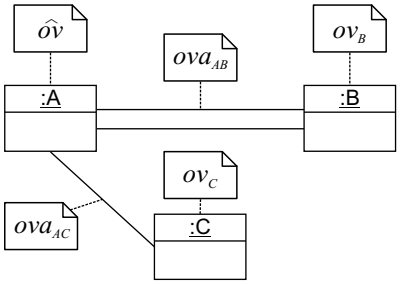

Figure 4.7: An example model variable mv_1 for *maxmatch*

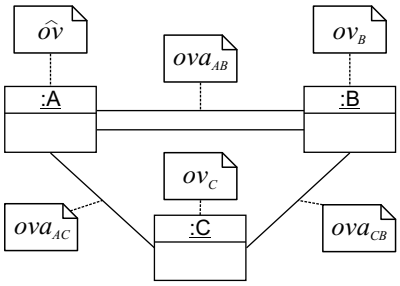

Figure 4.8: An example model variable mv_2 for *maxmatch*

Figures 4.7 and 4.8 show two model variables mv_1 and mv_2.

In the example we consider both times that $\widehat{ov}$ is the only fix object variable. In the first example it is easy to see that there are two possibilities for an object C and there is one possibility for an object B as this object has to be connected by at least two out of three associations (which are indistinguishable). So it holds:

$$maxmatch(mm, mv_1, \{\widehat{ov}\}) = 2 \cdot 1 = 2$$

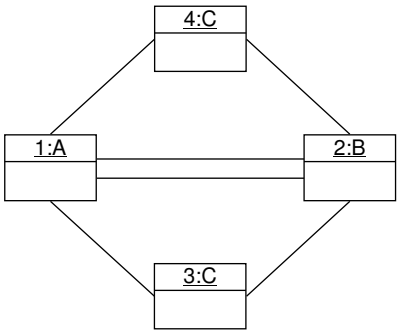

Figure 4.9: An example model fragment for *maxmatch*

In the example of Figure 4.8 the situation is slightly more complex but here it is not so easy to compute the result using combinatorics (as the graph is no longer a tree). But here again it holds:

$maxmatch(mm, mv_2, \{\widehat{ov}\}) = 2 \cdot 1 = 2$

In Figure 4.9 one example model fragment is shown which matches to mv_2 2 times.

Especially in cases where more than one object variable is considered as "fix" and those are not connected directly and the model variable is not simply a tree the calculation of *maxmatch* could be difficult. ❍

We now present the function *ubVarCard*. This function is used to to estimate how many outgoing associations might be maximal generated at the same object that stem from a given object variable association. Therefore the definition of *ubVarCard* uses the mapping *dependsOn* that helps to state how often the objects at the ends of the association might be the same resp. different.

Definition 4.2.17 (*ubVarCard*(ova, ae_1, r_i))

$ubVarCard(ova, ae_1, r_i) \mapsto n \in \mathbb{N}_0^\infty$ with $ova \in r_i|_{mv_1}|_{OVA} \wedge ae_1 \in ova|_{OVAT}$

Let ae_0, ov_0, ov_1 chosen so that

$$ae_0 = oppositeEnd(ova|_{OVAT}, ae_1) \tag{4.8}$$
$$ova|_{OVAE} = \{(ae_0, ov_0), (ae_1, ov_1)\} \tag{4.9}$$

Case 1: $ov_0 = ov_1$

See also Figure 4.10 and Figure 4.11 for the two possible cases for the association *ova*.

$ubVarCard(ova, ae_1, r_i) = ova|_{cardv}$

Case 2: $ov_0 \neq ov_1$

This situation is depicted in Figure 4.12. Table 4.2.17 shows the definition of *ubVarCard* for the sixteen different cases.

	$dependsOn(ov_0, r_i)$	$dependsOn(ov_1, r_i)$	$ubVarCard(ova, ae_1, r_i)$
(i)	$\emptyset$ (fixed)	$\emptyset$	$ova\|_{cardv}$
(ii)	$\perp$ (arbitrary)	$\emptyset$	$ova\|_{cardv}$
(iii)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\emptyset$	$ova\|_{cardv}$
(iv)	$\emptyset$	$\perp$	∞
(v)	$\perp$	$\perp$	∞
(vi)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\perp$	$maxmatch(r_i\|_{mv_0}\|_{mm}, r_i\|_{mv_0}, M_0) \cdot ova\|_{cardv}$
(vii)	$\emptyset$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	∞
$(viii)$	$\perp$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	∞
(ix)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\begin{cases} ova\|_{cardv} & \text{for } M_0 \overset{r_i\|_{mv_0}}{\rightsquigarrow} M_1 \\ maxmatch(r_i\|_{mv_0}\|_{mm}, r_i\|_{mv_0}, M_0) \cdot ova\|_{cardv} & \text{otherwise} \end{cases}$
(x)	$\diamondsuit$	$\emptyset$	$ova\|_{cardv}$
(xi)	$\diamondsuit$	$\perp$	$ova\|_{cardv}$
(xii)	$\diamondsuit$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
$(xiii)$	$\diamondsuit$	$\diamondsuit$	$ova\|_{cardv}$
(xiv)	$\emptyset$	$\diamondsuit$	∞
(xv)	$\perp$	$\diamondsuit$	∞
(xvi)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\diamondsuit$	$maxmatch(r_i\|_{mv_0}\|_{mm}, r_i\|_{mv_0}, M_0) \cdot ova\|_{cardv}$

Table 4.1: Definition of *ubVarCard*

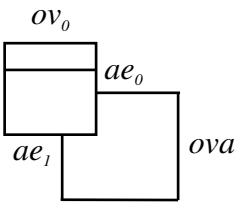

Figure 4.10: Reflexive association ($ov_0 = ov_1, ae_0 \neq ae_1$)

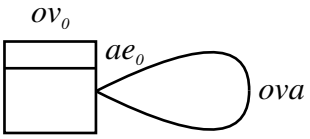

Figure 4.11: Symmetric, reflexive association ($ov_0 = ov_1, ae_0 = ae_1$)

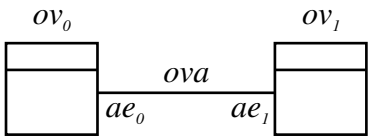

Figure 4.12: ov_0 and ov_1

❍

Lemma 4.2.4

$maxVarCard(ova, ae_1, r_i) \leq ubVarCard(ova, ae_1, r_i)$

❑

Proof: Let ae_0, ov_0, ov_1 be chosen as in Definition 4.2.17.

1. Case: $ov_0 = ov_1$

(*i*) $dependsOn(ov_0, r_i) = \emptyset$, i.e. $ov_0|_{oiv}$ is fix: The proof is directly evident from Lemma 3.4.4.

(*ii*) $dependsOn(ov_0, r_i) = \perp$, i.e. $ov_0|_{oiv}$ is free: The proof is directly evident from Lemma 3.4.4.

(*iii*) $dependsOn(ov_0, r_i) = \{ov'_i, ..., ov'_j\} \neq \emptyset$, i.e. the object variable identifier $ov_0|_{oiv}$ is one-to-one dependent on $\{ov'_i, ..., ov'_j\}$: The proof is directly evident from Lemma 3.4.4.

(*iv*) $dependsOn(ov_0, r_i) = \diamondsuit$ holds obviously because of the definition of $\diamondsuit$ and Lemma 3.4.4.

2. Case: $ov_0 \neq ov_1$

Let $m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$ be arbitrary.

(i-iii, x)

$$dependsOn(ov_1, r_i) = \emptyset \overset{\text{Def. } dependsOn}{\Rightarrow} ov_1|_{oiv} = const. \tag{a}$$
$$\overset{\text{Def. } mft,\ merge}{\Rightarrow} \exists_1 o' \in apply(m, r_i, (\emptyset,\emptyset))|_{OB} : mfm^1_\mu|_{match_o}(ov_1) = o'$$

i.e. exactly one object o' is created from ov_1.

$$\forall oa \in apply(m, r_i, (\emptyset,\emptyset))|_{OA} : (ae_1, mfm^1_\mu|_{match_o}(ov_1)) \in oa|_{AE} \tag{b}$$
$$\wedge\, mfm^1_\mu|_{match_a}(ova) = oa$$

i.e. all the associations that have been generated from one object variable association do have $(ae_1, mfm^1_\mu|_{match_o}(ov_1))$ as one end. Thus one end does contain an object o' that was created from the object variable ov_1.

$$\overset{\text{(a) and (b)}}{\Rightarrow} (ae_1, o') \text{ is an end of all associations that have been created from } ova$$
$$\Rightarrow \quad numAssos(r_i, m, o, ova, ae_1) \overset{\text{Def. 4.2.7}}{=}$$
$$= ova|_{cardv} \cdot \left|\{((ae_1, o'), (ae^0, o)), ..., ((ae_1, o'), (ae^n, o))\}\right|$$

Thereby it holds that $o' \neq o$, because

$$o = mfm^1_\mu|_{match_o}(ov_0) \wedge o' = mfm^1_\mu|_{match_o}(ov_1) \wedge ov_0 \neq ov_1$$

according to the assumption for 2. case.

It must hold that:

$$\forall oa \in \{(ae_1, o'), (ae^i, o^i)\} :$$
$$oa|_{AE} = \{(ae_1, o'), (oppositeEnd(ova|_{OVAT}, ae_1), o)\} \text{ (Def. 4.2.7)}$$
$$\Rightarrow \text{There exists maximal one element } ((ae_1, o'), (ae^i, o^i))$$

$$\forall o : o \in apply(m, r_i, (\emptyset,\emptyset))|_{OB} \wedge mfm^1_\mu|_{match_o}(ov_0) = o \text{ it holds:}$$
$$numAssos(r_i, m, o, ova, r_i) \leq ova|_{cardv}$$
$$\Rightarrow maxVarCard(ova, ae, r_i) \leq \max\{\max\{0, ova|_{cardv}\}\} = ova|_{cardv}$$
$$= ubVarCard(ova, ae_1, r_i)$$

(iv,v,vii,viii,xiv,xv) trivial because $\infty \geq maxVarCard(ova, ae_1, r_i)$.

(vi, xvi) $dependsOn(ov_0, r_i) = M_0 \neq \emptyset$

Substitution/rearranging of the definition of *numAssos* yields to

$$\begin{aligned}
&numAssos(r_i, m, o, ova, ae_1) \\
&\quad = ova|_{cardv} \cdot \left|\{\{(ae_0, o), (ae_1, o^0)\}, ..., \{(ae_0, o), (ae_1, o^n)\}\}\right| \\
&\quad \leq ova|_{cardv} \cdot (n+1) \qquad \text{(c)} \\
&\text{with } mfm^1_\mu|_{match_a}(ova) = oa = \{(ae_0, o), (ae_1, o^i)\} \\
&\qquad \text{whereby } (ae_0, o) = oppositeEnd(ova|_{OVAT}, ae_1)
\end{aligned}$$

$$\begin{aligned}
&\forall oa_i = \{(ae_0, o), (ae_1, o^i)\} : \exists mfm^1_\mu \in MFM^1 : \\
&\qquad mfm^1_\mu|_{match_a}(ova) = oa_i \wedge mfm^1_\mu|_{match_o}(ov_0) = o \qquad \text{(d)} \\
&\qquad \wedge\ mfm^1_\mu|_{match_o}(ov_1) = o^i
\end{aligned}$$

For every object association oa there exists a mfm^1_μ, which "generated" oa. The related association match $match_a(ova) = oa_i$ always maps to oa_i, while the object match $match_o(ov_1) = o^i$ for different associations oa_j, oa_i can also yield to the same object o, i.e. $mfm^1_{\mu_i}|_{match_o}(ov_1) = o = mfm^1_{\mu_j}|_{match_o}(ov_1)$

The set $\{\{(ae_0, o), (ae_1, o^0)\}, ..., \{(ae_0, o), (ae_1, o^n)\}\}$ has maximal as much elements as different o^i's may exist. I.e. it holds for $numAssos(r_i, m, o, ova, ae_1)$ from (c):

$$numAssos(r_i, m, o, ova, ae_1) = (n+1) \cdot ova|_{cardv}, o^i \neq o^j \forall 0 \leq i, j \leq n$$

From (d) follows that:

$$\begin{aligned}
&\left|\{oa_i : oa_i = \{(ae_0, o), (ae_1, o^i)\}\}\right| \overset{(d)}{\leq} n+1 \overset{\text{Def. } 3.4.10}{=} |mfm| \\
&dependsOn(ov_0, r_i) = M_0 \\
&\Rightarrow n+1 \leq maxmatch(r_i|_{mv_0}|_{mm}, r_i|_{mv_0}, M_0) \\
&\Rightarrow \max_{o \in apply(m, r_i, (\emptyset, \emptyset))|_{OB}} (numAssos(r_i, m, o, ova, ae_1)) = ova|_{cardv} \cdot (n+1) \\
&\qquad\qquad \leq ova|_{cardv} \cdot maxmatch(r_i|_{mv_0}|_{mm}, r_i|_{mv_0}, M_0)
\end{aligned}$$

Since m is fix but arbitrary it holds also that:

$$\begin{aligned}
&\max_{m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}} \Big(\max_{o \in apply(m, r_i, (\emptyset, \emptyset))|_{OB}} (numAssos(r_i, m, o, ova, ae_1)) \Big) \\
&\qquad\qquad \leq ova|_{cardv} \cdot maxmatch(r_i|_{mv_0}|_{mm}, r_i|_{mv_0}, M_0)
\end{aligned}$$

(ix) **1. Case:** generally it does always hold for $M_0 \neq \emptyset$ that:

$$maxVarCard(ova, ae_1, r_i) \leq ubVarCard(ova, ae_1, r_i)$$

and with

$$\emptyset \neq M_0 \subseteq r_i|_{mv_0}|_{OVB}$$

(c) holds surely. By $M_1 \neq \perp$ the set of o^i in (c) is further restricted.

$\Rightarrow$ Postulation

2. Case: $M_0 \overset{r_i|_{mv_0}}{\rightsquigarrow} M_1$

Let o' be an object that has been generated from the object variable ov_0. Because of $dependsOn(ov_0, r_i) = M_0$ we know that whenever the elements of M_0 match to the same source objects, the same target object o' is generated. Because of $dependsOn(ov_1, r_i) = M_1$ we know that whenever the elements of M_1 match to the same source objects, the same target object o'' is generated. Further $M_0 \overset{r_i|_{mv_0}}{\rightsquigarrow} M_1$ determines that whenever the elements of M_0 match to the same source objects, every element of M_1 matches deterministically to a unique source object, too. Thus we know that whenever o' is generated only the according object o'' is generated which is connected by an object association according to the type of ova.

(xi, xii, xiii)

$$\begin{aligned}
&dependsOn(ov_0, r_i) = \Diamond \\
&\Rightarrow \forall \mu, \tau : \mu \neq \tau \Rightarrow mfm^1_\mu|_{match_o}(ov_0) \neq mfm^1_\tau|_{match_o}(ov_0) \\
&\Rightarrow \text{ for arbitrary but fix } o, \mu \text{ with } mfm^1_\mu|_{match_o}(ov_0) = o \text{ it holds:} \\
&\quad \exists_1 mfm^1_\mu|_{match_a}(ova) = oa \text{ with } (ae_1, o) \in oa|_{OAE} \\
&\Rightarrow ubVarCard(ova, ae_1, r_i) = ova|_{cardv}
\end{aligned}$$

❑

Lemma 4.2.5
Let r be a rule set and $r_i, r_j \in r$ with $r_i|_{mv_0} \sqsubseteq r_j|_{mv_0}$ and *corresponds* a injective mapping according to Definition 4.2.14. Then it holds:

$$\begin{aligned}
&\forall m, mfm^j \text{ with } m \in \mathbb{M}_{mm} \wedge mfm^j \in MFM(m, r_j|_{mv_0}) : \\
&\quad \exists mfm^i \in MFM(m, r_i|_{mv_0}) : \\
&\qquad mfm^i|_{match_o} \circ corresponds = mfm^j|_{match_o}
\end{aligned}$$

❑

Proof Sketch: According to Definition 4.2.14 we know that $r_i|_{mv_0}$ is a substructure of $r_j|_{mv_0}$. Thus according to Definition 3.4.3 whenever a model fragment matches to $r_j|_{mv_0}$ then the according substructure of the model fragment matches to $r_i|_{mv_0}$. This means that corresponding object variables of the model variables match to the same objects. ❑

Definition 4.2.18 ($OVP(r_{super}, r_{sub})$)
Let $r_{super}, r_{sub} \in r$ be two rules for that it does hold $r_{sub}|_{mv_0} \sqsubseteq r_{super}|_{mv_0}$. According to Definition 4.2.14 there must exist a non-empty set of injective *corresponds* mappings that we denote with $\{corresponds_0, \ldots, corresponds_m\}$. Let $OVP_k(r_{super}, r_{sub})$ (with $k \in \{0, \ldots, m\}$) be the set of pairs of object variables $\{(ov^0_{super}, ov^0_{sub}), \ldots, (ov^n_{super}, ov^n_{sub})\}$ for that it holds:

(i) $\forall i \in \{0, \ldots, n\}$:

$ov^i_{super} \in r_{super}|_{mv_1|_{OVB}} \wedge$

$ov^i_{sub} \in r_{sub}|_{mv_1|_{OVB}} \wedge$

$ov^i_{super}|_{otv} = ov^i_{sub}|_{otv}$

(ii) For both rules r_{super} and r_{sub} the equational System *GL* according to Definition 3.4.5 has the same solution for all object identifiers of object variables that occur in OVP_k, i.e.

$\forall j \in \{0, \ldots, n\} : match^1_o(ov^i_{super})|_{id} = match^1_o(ov^i_{sub}))|_{id}[ov'/corresponds_k(ov')]_{ov' \in r_{sub}|_{mv_0}}$

Whereby $t(a)[a/b]$ denotes the term substitution that replaces any occurrence of a by b.

Further we denote $OVP(r_{super}, r_{sub}) = \{OVP_0(r_{super}, r_{sub}) \ldots OVP_m(r_{super}, r_{sub})\}$

If $r_{sub}|_{mv_0} \sqsubseteq r_{super}|_{mv_0}$ does not hold $OVP(r_{super}, r_{sub})$ yields to $\emptyset$. ❍

$OVP(r_{super}, r_{sub})$ yields to sets of tuples of object variables that always generate the same target objects. Please note that we do not make any statement about conflicting attribute values, since this property can be determined with the verification technique for applicability (c.f. Section 4.1).

Figure 4.13 shows an example where $r_{sub}|_{mv_0}$ is a substructure of $r_{super}|_{mv_0}$. A valid *corresponds* mapping between the two left hand structures is indicated by slashed arrows. Between the right hand model variables the tuples of the according *OVP* set is indicated.

Definition 4.2.19 ($redundant(ova_q, r_i, r)$)
Let r be a rule set with $r_i \in r$ and $ova_q \in r_i|_{mv_1}|_{OVA}$.

$redundant(ova_q, r_i, r) :\Leftrightarrow$

(i) $\exists r_p \in r : OVP(r_i, r_p) \neq \emptyset$, this implies $r_p|_{mv_0} \sqsubseteq r_i|_{mv_0} \wedge$

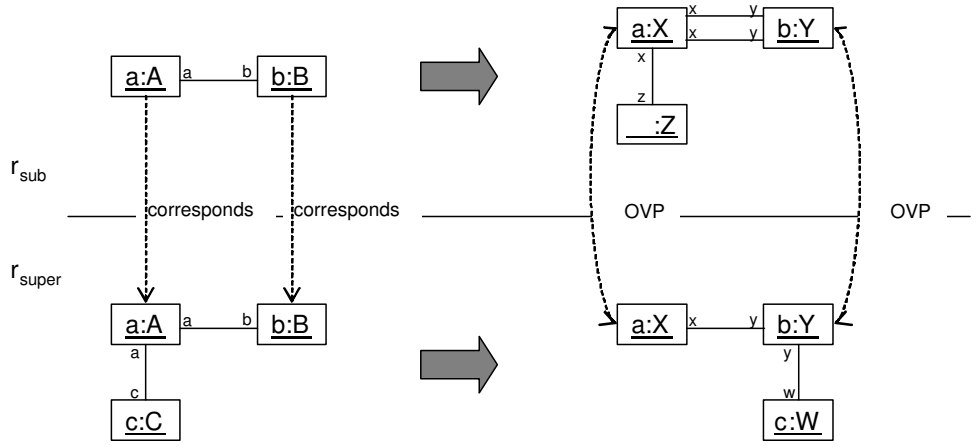

Figure 4.13: An example for $OVP(r_{super}, r_{sub})$

(ii) $\exists ova_q, cardv_j, OVAE_j, OVP_k(r_i, r_p)$:

$$OVP_k(r_i, r_p) \in OVP(r_i, r_p)$$
$$ova_q = (ova_q|_{OVAT}, cardv_j, OVAE_j) \in r_p|_{mv_1}|_{OVA} \wedge$$
$$OVAE_j = \{(ae, ov) : \exists (ae, ov_i) \in ova_q|_{OVAE} \wedge (ov_i, ov) \in OVP_k(r_i, r_p)\} \wedge$$
$$cardv_j \geq ova_q|_{cardv}$$

❍

For a given rule set r with rule a r_i *redundant* decides if a given right hand object variable association ova_q of r_i is redundant, i.e. all instances of this variable association will be created by at least one other rule. Thus *ova* has not to be considered when we estimate the upper bound of outgoing associations for generated models.

Lemma 4.2.6
Let be r an applicable rule set, $r_i \in r$, and $ova \in r_i|_{mv_1}|_{OVA}$. Then it holds:

$$redundant(ova, r_i, r) \Rightarrow maxRedundant(ova, r_i, r)$$

❑

Proof Sketch: Assumption: $redundant(ova_i, r_i, r)$ does hold.

Thus, according to Definition 4.2.19 there exists at least one rule r_j with $r_j \sqsubseteq r_i$. According to Lemma 4.2.5 we know that for a given but arbitrary model fragment *mf* which matches to $r_i|_{mv_0}$ the according substructure of *mf* matches also to $r_j|_{mv_0}$. Further according to Definition 4.2.18 there exist pairs of object variables in $r_i|_{mv_1}$ and $r_j|_{mv_1}$ which generate objects with identical identifiers (Def. 4.2.18 (ii)). As the rule set is applicable these objects are *mergeable*. Thus

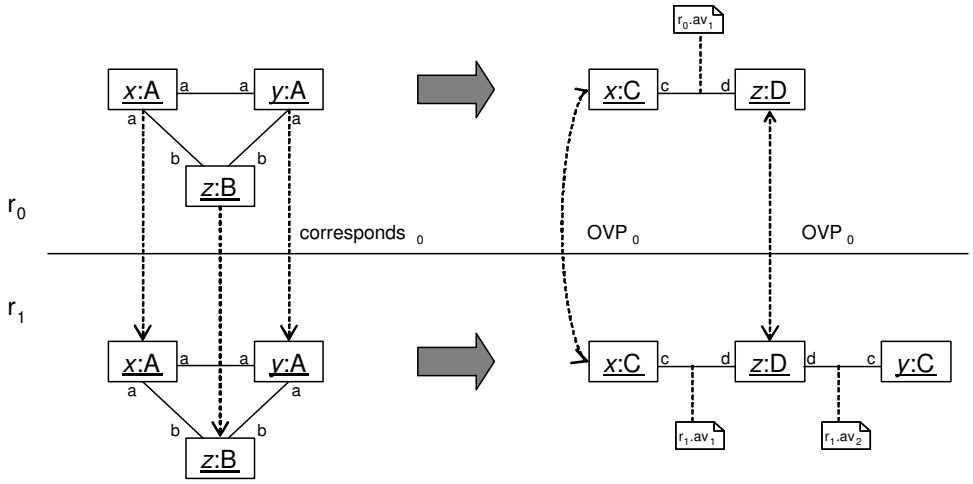

Figure 4.14: The second of two possible corresponds mappings between r_0 and r_1 and the according set *OVP*.

there are also object associations created by r_j which are superfluously created by r_i. Note that Definition 4.2.19 demands that the cardinalities of those object variable associations of r_j are at least as big as those in r_i. ❑

Example:

In this example two rules r_0 and r_1 are presented for which it holds that $r_0|_{mv_0} \cong r_1|_{mv_0}$. The rules are depicted in Figure 4.14 We can easily show that $r_0|_{mv_0} \sqsubseteq r_1|_{mv_0}$ holds according to Definition 4.2.14 by providing the injective mapping $corresponds_0$ as depicted at the left hand side of Figure 4.14. Further we can provide another injective mapping $corresponds_1$ as shown on the left side of Figure 4.15.

Therefore $OVP(r_1, r_0) = \{OVP_0(r_1, r_0), OVP_1(r_1, r_0)\}$ contains two sets of sets of tuples, one for each of the two possible *corresponds* mappings. The first one OVP_0 is shown at the right hand side of Figure 4.14, the second possible set of tuples OVP_1 is shown at the right hand side of Figure 4.15.

Thus we can calculate *redundant* for r_1's right hand association variables regarding the rule set $r = (r_0, r_1)$. Definition 4.2.19 (i) holds obviously, since we know that $OVP(r_1, r_0) \neq \emptyset$.

For $redundant(r_1.av_1, r_1, r)$ we can see that the an association variable $r_0.av_1$ of r_0 has the properties required by Definition 4.2.19 (ii) when we regard $OVP_0(r_1, r_0)$ as the $OVP_k(r_1, r_0)$ from Definition 4.2.19 (ii).

To show that $redundant(r_1.av_2, r_1, r)$ also holds we must consider $OVP_0(r_1, r_0)$ as the $OVP_k(r_1, r_0)$ from Definition 4.2.19 (ii).

Thus both association variables in r_1 are redundant, i.e. they would be created by another rule (in this case the rule r_0) anyway. However we can easily show that the association variable $r_0.av_1$ in

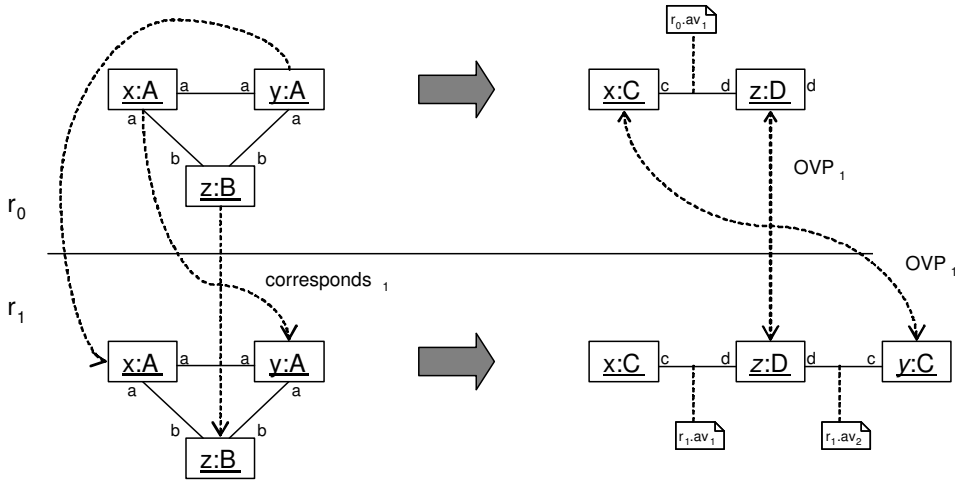

Figure 4.15: One of two possible corresponds mappings between r_0 and r_1 and the according set *OVP*.

rule r_0 is redundant, too. This is the case because r_0 and r_1 create exactly the same associations.

❍

Definition 4.2.20 (*relevant*$(r, r_i, AE, ae_1, ova', ov_0, ov_0')$)

$$\begin{aligned} relevant(r, r_i, AE, ae_1, ova', ov_0, ov_0') :\Leftrightarrow\ & r_i \in r \\ & \wedge\, ova' \in r_i|_{mv_1}|_{OVA} \\ & \wedge\, ova'|_{OVAE}|_{ae} = AE \\ & \wedge\, (oppositeEnd(AE, ae_1), ov_0') \in ova'|_{OVAE} \\ & \wedge\, ov_0'|_{otv} = ov_0|_{otv} \\ & \wedge\, ov_0'|_{oiv} \sim ov_0|_{oiv} \\ & \wedge\, \neg\big(redundant(ova', r_i, r) \\ & \qquad \wedge\, \nexists r_j : (\, j < i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \\ & \qquad\qquad \wedge\, redundant(ova', r_i, (r_j, r_i)))\big) \end{aligned}$$

❍

Similar to *maxRelevant* the predicate *relevant* decides, if an outgoing object variable association is relevant in the context of a given rule set r. This is the case if we may not neglect this association without changing the possible results of a model transformation. While *maxRelevant* yields to optimal answers *relevant* uses an easy computable estimation for the detection of redundant object variable associations.

relevant gets a rule set r and a rule r_i as a context. Further the type of the association we look at is specified by a class association *AE*, a class association end ae_1 and an object variable ov_0.

This triple specifies an outgoing object variable association starting at ov_0 with the "opposite" end ae_1.

relevant compares this specification for similar object variable associations. If the following is all true, then the outgoing object variable association specified by ova' and ov'_0 is relevant:

(*i*) ova' and ov'_0 are part of the right hand side of the rule r_i.

(*ii*) The object variable association has the according type AE.

(*iii*) The object variable ov'_0 is connected to ova' at the opposite side of ae_1.

(*iv*) The type of ov'_0 is the same then the type of ov_0 and both object variables are similar (i.e. have similar identifier terms).

(*v*) We do not consider the given object variable association, if it is redundant and there exists no rule r_j with $j < i$ which has a congruent left hand model variable and to which r_i is redundant. In this special case the association variables in the two rules would be mutual redundant and we make the policy to count only those in the rule with the smaller index.

Theorem 4.2.2 (Verification technique for upper bounds conformance)
Let r be a rule set with $mm_1 = r_0|_{mv_1}|_{mm}$, $CA_1 = mm_1|_{CA}$. The rule set r generates an upper bounds conform model fragment, if

(*i*) $\forall r_k, ov_0, ae_1, AE, ub$ with $r_k \in r$,
$AE =\in CA_1$,
$ae_1 \in AE$,
$(oppositeEnd(AE, ae_1), ov_0) \in r|_{mv_1}|_{OVA}|_{OVAE}$,
$(lb, ub) = ae_1|_m$:

$$ub \geq \begin{cases} \sum\limits_{\substack{ova':ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae} = AE \\ \wedge (oppositeEnd(AE,ae_1),ov_0) \in ova'|_{OVAE}}} ova'|_{cardv} & \text{if } ov_0|_{oiv} = \diamondsuit \\ \sum\limits_{\substack{r_i, ova', ov'_0: \\ relevant(r,r_i,AE,ae_1,ova',ov_0,ov'_0)}} ubVarCard(ova', ae_1, r_i) & \text{otherwise} \end{cases}$$

(*ii*) r is applicable.

❑

The verification technique for upper bounds conformance adds up some estimation for the cardinalities of all outgoing associations. If these sums are less or equal to the according upper bounds and the rule set is applicable then the rule set is upper bounds conform.

The estimation is calculated for every rule and every possible outgoing association. Thereby two cases are distinguished. If the association starts at an object variable with the identifier term $\diamondsuit$

then only the cardinalities of all outgoing associations of the same type of this object variable have to be added. Because of the $\diamondsuit$ value it is ensured that each object generated by the object variable is unique and never touched by another rule application.

In the second case all other rules, object variable associations, and object variables have to be considered. Then the sum of the relevant association have to be calculated.

Proof: *Case I:* $ov_0|_{oiv} = \diamondsuit$

$$
\begin{aligned}
& ub \\
\geq \quad & \sum_{\substack{ova':ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae}=AE \\ \wedge(oppositeEnd(AE,ae_1),ov_0) \in ova'|_{OVAE}}} ova'|_{cardv} \\
\text{Tab. 4.2.17 } (x)\text{–}(xiii) \quad = \quad & \sum_{\substack{ova':ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae}=AE \\ \wedge(oppositeEnd(AE,ae_1),ov_0) \in ova'|_{OVAE}}} ubVarCard(ova', ae_1, r_k) \\
\text{Lemma 4.2.4} \quad \geq \quad & \sum_{\substack{ova':ova' \in r_k|_{mv_1}|_{OVA} \\ \wedge ova'|_{OVAE}|_{ae}=AE \\ \wedge(oppositeEnd(AE,ae_1),ov_0) \in ova'|_{OVAE}}} maxVarCard(ova', ae_1, r_k) \\
\text{Lemma 4.2.3} \quad \geq \quad & maxCard(AE, ae_1, r)
\end{aligned}
$$

Case II: $ov_0|_{oiv} \neq \diamondsuit$

We show in the first subsidiary that the number of summands of the sum in Theorem 4.2.2 is at least as big as the number of summands in Lemma 4.2.3. Together with the fact that *ubVarCard* is bigger or equal than *maxVarCard* we know that the first sum is bigger.

Subsidiary:

$$
\begin{aligned}
& relevant(r, r_i, AE, ae_1, ova', ov_0, ov_0') \\
& \Rightarrow maxRelevant(r, r_i, AE, ae_1, ova', ov_0, ov_0') \\
& \quad \equiv \\
& \big(r_i \in r \wedge ova' \in r_i|_{mv_1}|_{OVA} \wedge ova'|_{OVAE}|_{ae} = AE \\
& \quad \wedge (oppositeEnd(AE, ae_1), ov_0') \in ova'|_{OVAE} \wedge ov_0'|_{otv} = ov_0|_{otv} \wedge ov_0'|_{oiv} \sim ov_0|_{oiv} \\
& \quad \wedge maxRedundant(ova, r_i, r) \wedge \nexists r_j : \big(j < i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(ova', r_i, (r_j, r_i))\big)\big) \\
& \Rightarrow \big(r_i \in r \wedge ova' \in r_i|_{mv_1}|_{OVA} \wedge ova'|_{OVAE}|_{ae} = AE \\
& \quad \wedge (oppositeEnd(AE, ae_1), ov_0') \in ova'|_{OVAE} \wedge ov_0'|_{otv} = ov_0|_{otv} \wedge ov_0'|_{oiv} \sim ov_0|_{oiv} \\
& \quad \wedge redundant(ova, r_i, r) \wedge \nexists r_j : \big(j < i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(ova', r_i, (r_j, r_i))\big)\big) \\
& \quad \equiv
\end{aligned}
$$

$$
\begin{aligned}
&\big(r_i \in r \wedge ova' \in r_i|_{mv_1}|_{OVA} \wedge ova'|_{OVAE}|_{ae} = AE \\
&\quad \wedge (oppositeEnd(AE, ae_1), ov'_0) \in ova'|_{OVAE} \wedge ov'_0|_{otv} = ov_0|_{otv} \wedge ov'_0|_{oiv} \sim ov_0|_{oiv} \\
&\quad \wedge \big(\neg maxRedundant(ova, r_i, r) \vee \\
&\qquad \big(\neg \forall r_j : (j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg maxRedundant(ova, r_i, (r_j, r_i)))\big)\big) \\
&\Rightarrow \big(r_i \in r \wedge ova' \in r_i|_{mv_1}|_{OVA} \wedge ova'|_{OVAE}|_{ae} = AE \\
&\qquad \wedge (oppositeEnd(AE, ae_1), ov'_0) \in ova'|_{OVAE} \wedge ov'_0|_{otv} = ov_0|_{otv} \wedge ov'_0|_{oiv} \sim ov_0|_{oiv} \\
&\qquad \wedge \big(\neg redundant(ova, r_i, r) \vee \big(\neg \forall r_j : (j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big)\big) \\
&\quad \equiv \\
&\big(\neg maxRedundant(ova, r_i, r) \vee \\
&\quad \big(\neg \forall r_j : (j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg maxRedundant(ova, r_i, (r_j, r_i)))\big)\big) \\
&\Rightarrow \big(\neg redundant(ova, r_i, r) \vee \big(\neg \forall r_j : (j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big) \\
&\quad \equiv \\
&\neg \forall r_j : \\
&\big(\neg maxRedundant(ova, r_i, r) \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg maxRedundant(ova, r_i, (r_j, r_i)))\big)\big) \\
&\Rightarrow \big(\neg redundant(ova, r_i, r) \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big) \\
&\quad \equiv \\
&\big(maxRedundant(ova, r_i, r) \wedge (j < i) \wedge (r_j|_{mv_0} \cong r_i|_{mv_0}) \wedge maxRedundant(ova, r_i, (r_j, r_i))\big) \\
&\vee \big(\neg redundant(ova, r_i, r) \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big) \\
&\quad \equiv \\
&\Big(maxRedundant(ova, r_i, r) \vee \big(\neg redundant(ova, r_i, r) \\
&\qquad \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big)\Big) \\
&\wedge \Big(j < i \vee \big(\neg redundant(ova, r_i, r) \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big)\Big) \\
&\wedge \Big(r_j|_{mv_0} \cong r_i|_{mv_0} \vee \big(\neg redundant(ova, r_i, r) \\
&\quad \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big)\Big) \\
&\wedge \Big(maxRedundant(ova, r_i, (r_j, r_i))\big) \vee \big(\neg redundant(ova, r_i, r) \\
&\qquad \vee \big((j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\big)\big)\Big) \\
&\quad \equiv \\
&\Big((maxRedundant(ova, r_i, r) \vee \neg redundant(ova, r_i, r)) \\
&\qquad \vee (j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i)))\Big) \\
&\wedge \Big((j < i \vee j \geq i) \vee \neg redundant(ova, r_i, r) \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg redundant(ova, r_i, (r_j, r_i))\Big)
\end{aligned}
$$

$$
\begin{array}{l}
maxCard(AE, ae_1, r) \\
\leq \qquad \text{(Lemma 4.2.3)} \\
\overbrace{\underbrace{\sum_{\diamondsuit} maxVarCard(ova, ae_1, r_i)}_{} \qquad \underbrace{\sum_{maxRelevant} maxVarCard(ova, ae_1, r_i)}_{}}^{} \\
\leq \qquad \text{(Lemma 4.2.4)} \\
\overbrace{\underbrace{\sum_{\diamondsuit} ova|_{cardv}}_{} \qquad \underbrace{\sum_{relevant} ubVarCard(ova, ae_1, r_i)}_{}}^{} \qquad \text{(Verif. Techn.)} \\
\leq \qquad \text{(Theorem 4.2.2)} \\
ub
\end{array}
$$

Figure 4.16: Proof chain for the verification technique for upper bounds conformance

$$
\begin{array}{l}
\wedge\Big((r_j|_{mv_0} \cong r_i|_{mv_0} \vee r_j|_{mv_0} \ncong r_i|_{mv_0}) \vee \neg redundant(ova, r_i, r) \vee j \geq i \\
\qquad \vee \neg redundant(ova, r_i, (r_j, r_i)) \Big) \\
\wedge\Big((maxRedundant(ova, r_i, (r_j, r_i)) \vee \\
\quad \neg redundant(ova, r_i, (r_j, r_i))) \vee \neg redundant(ova, r_i, r) \\
\qquad \vee j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \Big) \\
\quad \underset{\equiv}{redundant \Rightarrow maxRedundant} \\
true
\end{array}
$$

$$
\begin{array}{l}
ub_j \geq \sum_{\substack{r_i, ova: \; r_i \in r, \\ ova \in r_i|_{mv_1}|_{OVA}}} ubVarCard(ova, ae_j, r_i) \overset{\text{Subsidiary and Lemma 4.2.4}}{\geq} \\
\qquad \wedge\Big(\neg redundant(ova, r_i, r) \\
\qquad \vee\Big(\forall r_j : (j > i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \\
\qquad \vee \neg redundant(ova, r_i, (r_j))) \Big) \Big) \\
\qquad \sum_{r_i, ova: \; , r_i \in r,} maxVarCard(ova, ae, r_i) \geq maxCard(AE, ae, r) \\
\wedge\Big(\neg maxRedundant(ova, r_i, r) \\
\quad \vee\Big(\forall r_j : (j > i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \\
\vee \neg maxRedundant(ova, r_i, (r_j))) \Big) \Big) \\
\Rightarrow ubConform(r) \qquad \square
\end{array}
$$

Figure 4.16 shows the equation chain which was proven within this section in an informal way. The verification technique demands that the sums of the estimated number of created object

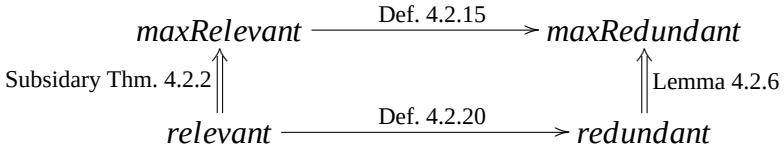

Figure 4.17: Dependencies between *relevant*, *redundant*, and their ideal relatives

associations is less or equal to the according upper bound. Thereby two cases are distinguished. If an (outgoing) object association is created by an object association variable connected to an object variable with an identifier equal to $\diamondsuit$ then only all other outgoing cardinalities of outgoing object variable associations of the same type have to be added. Otherwise the whole rule set has to be searched for other similar object variable associations which are relevant. Both sums are bigger or equal to the "ideal" estimations which build upon *maxVarCard* and *maxRelevant*. Those sums again are alway less or equal to the potential maximal cardinalities of an association which is represented by *maxCard*. So this verification technique can ensure the upper bounds conformance of a given rule set r because the cardinalities of every association in a model is less or equal to the defined upper bound in its metamodel.

In Figure 4.17 the dependencies between *relevant*, *redundant*, and their relatives are depicted. For practical applicability the results of the "ideal" predicates *maxRelevant* and *maxRedundant* have to be estimated by some easy computable predicates *relevant* and *redundant*. Both ensure by construction that they imply their "ideal" relatives.

4.2.3 Verification Techniques for lower bounds conformance

Definition 4.2.21 (Lower bounds conform rules / rule sets (*lbConform*(r), *lbConform*(r_i)))
A rule set r with a source meta model $mm_0 = r_0|_{mv_0}|_{mm}$ and target meta model $mm_1 = r_0|_{mv_1}|_{mm}$ is called *lower bounds conform,* if it holds:

$\forall m \in \mathbb{M}_{mm_0} : lbConform(transform(m,r), mm_1)$

For short we write: $lbConform(r)$.

A rule r_i is *lower bounds conform,* i.e. $lbConform(r_i)$ holds, if

$\forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : lbConform(apply(m, r_i, (\emptyset,\emptyset)), r_i|_{mv_1}|_{mm}))$

$\bigcirc$

Definition 4.2.22 (*varLbConform*(mv,mm))
varLbConform(mv,mm) is a predicate so that it holds

$$
\begin{aligned}
&varLbConform(mv,mm)\\
&\quad :\Leftrightarrow\\
&\forall AE \in mm|_{CA} : \forall ae \in AE : (lb,ub) = oppositeEnd(AE,ae)|_m :\\
&\qquad \forall ov \in mv|_{OVB} \text{ with } ov|_{otv} = ae|_c : lb \leq \sum_{\substack{ova \in mv|_{OVA} \text{ with}\\ ova|_{ovat}=AE\\ \wedge (ae,ov)\in ova|_{OVAE}}} ova|_{cardv}
\end{aligned}
$$

❍

varLbConform(mv,mm) holds, if and only if, the sum of all cardinalities of out-going object variable associations for every object variable in a model variable *mv* are equal or above the lower bounds of the regarding multiplicities in the metamodel *mm*.

Lemma 4.2.7
For every model fragment *mf* that was generated by a rule application

$$apply(m,r_i,mf') = mf$$

from a model fragment mf' and a rule r_i it holds with regard to the target metamodel $mm = r_i|_{mv_1}|_{mm}$:

$$lbConform(mf',mm) \wedge varLbConform(r_i|_{mv_1},mm) \Rightarrow lbConform(mf,mm)$$

❑

That means if the target model variable of a rule is *varLbConform* to the target metamodel, than the result of the application of this rule is *lbConform* on the assumption that the previous model fragment mf' is *lbConform*.

Proof: Assumption: $mm = r_i|_{mv_1}|_{mm}$,
$varLbConform(r_i|mv_1,mm)$,
$lbConform(mf',mm)$

$$
\begin{aligned}
&apply(m,r_i,mf')\\
&= mf' \cup_{\mathfrak{m}} mft(mfm_0,r_i) \cup_{\mathfrak{m}} \cdots \cup_{\mathfrak{m}} mft(mfm_{|mfm|-1},r_i)\\
&= mf' \cup_{\mathfrak{m}} ((\emptyset,\emptyset) \cup_{\mathfrak{m}} mft(mfm_0,r_i) \cup_{\mathfrak{m}} \cdots \cup_{\mathfrak{m}} mft(mfm_{|mfm|-1},r_i))\\
&= mf' \cup_{\mathfrak{m}} apply(m,r_i,(\emptyset,\emptyset))
\end{aligned}
$$

(i) Assump.: $lbConform(\overline{mf}, mm)$ with $\overline{mf} = apply(m, r_i, (\emptyset, \emptyset))$

Proof: Let $(o \in \overline{mf}|_{OB}, AE : o|_{ot} \in AE|_c, ae \in AE)$ be arbitrary but fix. Because of $varLbConform(r_i|_{mv_1}, mm)$ together with $(lb, ub) = oppositeEnd(AE, ae)|_m$ it holds that:

According to Lemma 4.2.2 it holds:

$$\exists \overline{OV} : \sum_{\substack{oae:oae=(ae,o) \\ \wedge oae \in oa|_{OAE} \\ \wedge oa|_{OAT}=AE}} oa|_{card} \overset{\text{Lemma 4.2.2}}{\geq} \min_{ov \in \overline{OV}} \Big(\sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} \Big)$$

$$\overset{\min_A \geq \min_B \text{ for } A \subseteq B}{\geq} \min_{\substack{ov \in r_i|_{mv_1}|_{OVB} \\ \wedge ov|_{otv}=ae|_c}} \Big(\sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} \Big)$$

$$\overset{varLbConform(r_i|_{mv_1}, mm)}{\geq} lb$$

$\Rightarrow lbConform(\overline{mf}, mm)$

(ii) Assump.: $lbConform(mf_0) \wedge lbConform(mf_1) \Rightarrow lbConform(mf_0 \cup_{\mathfrak{m}} mf_1, mm)$

Proof: According to Lemma 3.4.4 (i)

$\overset{(i) \text{ and } (ii)}{\Rightarrow}$ Lemma 4.2.7 ❏

Theorem 4.2.3 (Simple verification technique for lower bounds conformance)
For a ruleset r with a metamodel mm it holds:

$\big(\forall r_i|_{mv_1} : varLbConform(r_i|_{mv_1}, mm)\big) \wedge$
r is applicable
$\Rightarrow lbConform(r)$

❏

This means that if the target model variables of all rules in an applicable rule set are *varLbConform*, then the result of the transformation of an arbitrary model according to this rule set is *lbConform*.

Proof: Proof by induction over i; let $mm_1 = r_1|_{mv_1}|_{mm}$

Induction hypothesis: $mf_0 = (\emptyset, \emptyset)$ (cf. Def. 3.4.11) $\Rightarrow lbConform(mf_0, mm_1)$

Induction step: $mf_i = apply(m, r_{i-1}, mf_{i-1})$

$lbConform(mf_{i-1}, mm_1)$ according induction precondition
$varLbConform(r_{i-1}|_{mv_1}, mm_1)$ according to the precondition
$\overset{\text{Lemma 4.2.7}}{\Rightarrow} lbConform(apply(m, r_{i-1}, mf_{i-1}))$

❑

Informally spoken one can prove the lower bounds conformance of a rule set by applying the following steps:

1. For every object variable in every rule remember the lower bounds of its outgoing class associations.
2. Check for every object variable in every rule that the sum of the outgoing object variable associations of each given association type is greater or equal than the lower bound of the class association found in step (1).

Thus a rule designer can easily determine whether his rules are lower bounds conform, by verifying that for every object variable the minimal required outgoing associations do exist. However, in specific cases this technique may be not good enough. Therefore we provide a more sophisticated verification technique within the rest of this section.

Lemma 4.2.8
For a rule r_i with $mv = r_i|_{mv_1}$ and $mm = r_i|_{mv_1}|_{mm}$ it holds:

$varLbConform(mv, mm) \Rightarrow lbConform(r_i)$

❑

Proof: The proof is evident as Theorem 4.2.3 holds also for rule sets that consist only of one rule. ❑

Definition 4.2.23 (*minCard*(AE, ae_1, r_i))

$minCard(AE, ae_1, r_i) \mapsto n \in \mathbb{N}_0^\infty$ with $ae_1 \in AE$

$$minCard(AE, ae_1, r_i) = \min_{m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}} \left(\min_{o \in apply(m, r_i, (\emptyset,\emptyset))|_{OB}} \left(\sum_{ova : ova|_{OVAT} = AE} numAssos(r_i, m, o, ova, ae_1) \right) \right)$$

❍

Lemma 4.2.9

Let $m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$ arbitrary and $mf = apply(m, r_i, (\emptyset, \emptyset))$

It does hold:

$$\forall o \in mf|_{OB} : \sum_{\substack{oa \in mf|_{OA}: \\ oa|_{oat} = AE \\ \wedge (ae_1, o) \in oa|_{OAE}}} oa|_{card} \geq minCard(AE, ae_1, r_i) \text{ with } ae_1 \in AE, AE \in r_i|_{mv_1}|_{mm}|_{CA}$$

❑

This means the mapping *minCard* yields to for arbitrary source models the potential minimal number of associations of the type *AE* within the target model. Thus *minCard* can be used to estimate how many instances of associations are at least generated ending at an object. So *minCard* helps to decide weather or not multiplicity constraints of the target metamodel might be violated in the generated model fragment.

Proof: Lemma 4.2.9 is directly evident from Definition 4.2.23. ❑

Definition 4.2.24 ($minVarCard(ova, ae_1, r_i)$**)**

$$minVarCard(ova, ae_1, r_i) = \min_{m \in \mathbb{M}_{r_i|mv_0|mm}} (\min_{o \in apply(m, r_i, (\emptyset,\emptyset))|_{OB}} (numAssos(r_i, m, o, ova, ae_1)))$$

❍

minVarCard yields to the minimum possible number of associations starting at an object that have been created from the same object variable association *ova*. This holds for an arbitrary source model.

Lemma 4.2.10

$$minCard(AE, ae_1, r_i) \geq \min_{ova:ova|_{OVAT}=AE} (minVarCard(ova, ae_1, r_i))$$

❑

Proof Sketch: Generally it holds that:

$$\min_k \Big(\sum_l f(k,l) \Big) \geq \min_{k,l} \Big(f(k,l) \Big)$$

Substitution into the definition yields to the proposition. ❑

Definition 4.2.25 ($minmatch(mm, mv, \{ov_i, ..., ov_j\})$**)**

$$minmatch(mm, mv, \{ov_i, ..., ov_j\}) \mapsto \mathbb{N}_0^\infty \text{ with } \{ov_i, ..., ov_j\} \subseteq mv|_{OVB}$$

For a metamodel *mm*, a model variable *mv*, and a set of object variables that are in *mv* and marked as "fix" *minmatch* yields to the minimum number of possible matches of the model variable in an arbitrary model, where the fix object variables are regarded as arbitrary but fix and existing.

❍

Note: For the following special cases it holds:

$minmatch(mm, mv, \perp) = 0$
$minmatch(mm, mv, \emptyset) = 0$

We now present the function *lbVarCard*. This function is used to to estimate how many outgoing associations might be at least generated at the same object that stem from a given object variable association. Therefore the definition of *lbVarCard* uses the mapping *dependsOn* that helps to state how often the objects at the ends of the association might be the same resp. different.

Definition 4.2.26 (*lbVarCard*(ova, ae_1, r_i))

$lbVarCard(ova, ae_1, r_i) \mapsto n \in \mathbb{N}_0^\infty$ with $ova \in r_i|_{mv_1}|_{OVA} \wedge ae_1 \in ova|_{OVAT}$

Let ae_0, ov_0, ov_1 chosen so that

$$ae_0 = oppositeEnd(ova|_{OVAT}, ae_1) \tag{4.10}$$
$$ova|_{OVAE} = \{(ae_0, ov_0), (ae_1, ov_1)\} \tag{4.11}$$

Case 1: $ov_0 = ov_1$

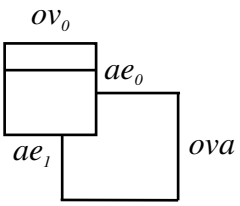

Figure 4.18: Reflexive association ($ov_0 = ov_1, ae_0 \neq ae_1$)

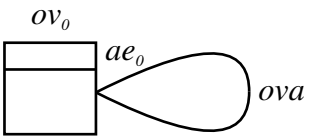

Figure 4.19: Symmetric, reflexive association ($ov_0 = ov_1, ae_0 = ae_1$)

See also Figure 4.18 and Figure 4.19 for the two possible cases for the association *ova*.

$lbVarCard(ova, ae_1, r_i) = ova|_{cardv}$

Case 2: $ov_0 \neq ov_1$

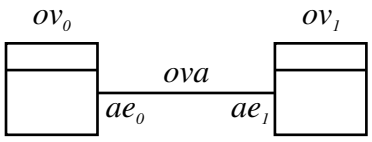

Figure 4.20: ov_0 and ov_1

The situation is depicted in Figure 4.20. Table 4.2.26 shows the definition of *lbVarCard* for the sixteen different cases.

	$dependsOn(ov_0, r_i)$	$dependsOn(ov_1, r_i)$	$lbVarCard(ova, ae_1, r_i)$
(i)	$\emptyset$ (fixed)	$\emptyset$	$ova\|_{cardv}$
(ii)	$\perp$ (arbitrary)	$\emptyset$	$ova\|_{cardv}$
(iii)	$M_0 \neq \emptyset$	$\emptyset$	$ova\|_{cardv}$
(iv)	$\emptyset$	$\perp$	∞
(v)	$\perp$	$\perp$	∞
(vi)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\perp$	$ova\|_{cardv}$
(vii)	$\emptyset$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	$minmatch(r_i\|_{mv_0}\|_{mm}, r_i\|_{mv_0}, M_0) \cdot ova\|_{cardv}$
$(viii)$	$\perp$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
(ix)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
(x)	$\diamondsuit$	$\emptyset$	$ova\|_{cardv}$
(xi)	$\diamondsuit$	$\perp$	$ova\|_{cardv}$
(xii)	$\diamondsuit$	$\emptyset \neq M_1 \subseteq r_i\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
$(xiii)$	$\diamondsuit$	$\diamondsuit$	$ova\|_{cardv}$
(xiv)	$\emptyset$	$\diamondsuit$	$ova\|_{cardv}$
(xv)	$\perp$	$\diamondsuit$	$ova\|_{cardv}$
(xvi)	$\emptyset \neq M_0 \subseteq r_i\|_{mv_0}\|_{OVB}$	$\diamondsuit$	$minmatch(r_i\|_{mv_0}\|_{mm}, r_i\|_{mv_0}, M_0) \cdot ova\|_{cardv}$

Table 4.2: Definition of *lbVarCard*

❍

Lemma 4.2.11

$$minVarCard(ova, ae_1, r_i) \geq lbVarCard(ova, ae_1, r_i)$$

❑

The proof of Lemma 4.2.11 is skipped here to safe space. Generally it is very similar to the considerations made in the proof of Lemma 4.2.4.

Definition 4.2.27 ($lbCard(AE, ae_1, r_i)$)

$$lbCard(AE, ae_1, r_i) \mapsto n \in \mathbb{N}_0^\infty, \text{ with } ae_1 \in AE$$

$$lbCard(AE, ae_1, r_i) = \begin{cases} 0 & \text{if } \exists ov \in r_i|_{mv_1}|_{OVB} \\ & \quad \wedge ov|_{otv} = oppositeEnd(AE, ae_1)|_c : \\ & \quad\quad \nexists ova \in r_i|_{mv_1}|_{OVA} : \\ & \quad\quad\quad (oppositeEnd(AE, ae_1), ov) \in ova|_{OVAE} \\ \min\limits_{ova:ova|_{OVAT}=AE}(lbVarCard(ova, ae_1, r_i)) & \text{otherwise} \end{cases}$$

❍

The first case of Definition 4.2.27 describes the case when we find an object variable *ov* that should have outgoing associations of the type *AE* that do not exist in the right hand model variable $r_i|_{mv_1}$.

Lemma 4.2.12

$$minCard(AE, ae_1, r_i) \geq lbCard(AE, ae_1, r_i)$$

❑

Proof:

$$\begin{aligned} minCard(AE, ae_1, r_i) &\overset{\text{Lemma 4.2.10}}{\geq} \min_{ova:ova|_{OVAT}=AE} minVarCard(ova, ae_1, r_i) \\ &\overset{\text{Lemma 4.2.11}}{\geq} \min_{ova:ova|_{OVAT}=AE} lbVarCard(ova, ae_1, r_i) \\ &\overset{\text{Def. 4.2.27}}{=} lbCard(AE, ae_1, r_i) \end{aligned}$$

❑

Theorem 4.2.4 (Verification Technique for the lbConformance of a rule)
Let r_i be a rule with a right hand model variable $mv = r_i|_{mv_1}$ and an according target metamodel $mm = mv|_{mm}$. It holds $lbConform(r_i)$, if

$\forall AE, ae$ with $AE \in mm|_{CA}, ae \in AE \wedge ae|_c \in mv|_{OVB}|_{otv}$:
$lbCard(AE, ae, r_i) \geq lb,$ with $ae|_m = (lb, ub)$

❑

Proof: Let $m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$ arbitrary and $mf = apply(m, r_i, (\emptyset, \emptyset))$. It does hold:

$\forall o \in mf|_{OB}$:

$$\sum_{\substack{oa \in mf|_{OA}: \\ oa|_{oat} = AE \\ \wedge (ae_1, o) \in oa|_{OAE}}} oa|_{card} \overset{\text{Lemma 4.2.9}}{\geq} minCard(AE, ae, r_i) \overset{\text{Lemma 4.2.12}}{\geq} lbCard(AE, ae, r_i) \geq lb$$

$$\overset{\text{Def. 4.2.5}}{\Rightarrow} lbConform(r_i)$$

❑

$$\begin{aligned}
& minCard(AE, ae_1, r_i) \\
\geq \quad & \text{(Lemma 4.2.10)} \\
& \min_{ova}(minVarCard(ova, ae_1, r_i)) \\
\geq \quad & \text{(Lemma 4.2.11)} \\
lbCard(AE, ae_1, r_i) = & \min_{ova}(lbVarCard(ova, ae_1, r_i)) \quad \text{(Verif. Techn.)} \\
\geq \quad & \text{(Theorem 4.2.5)} \\
& lb
\end{aligned}$$

Figure 4.21: Proof chain for the verification technique for lower bounds conformance

Figure 4.21 shows informally the proof chain used for the proof of the enhanced verification technique for lower bounds conformance. Whereas *minCard* yields to the minimal possible number of generated associations for a given rule and a given association type, the minimum of all *minVarCard*s is already an ideal estimation. An easily computable estimation for *minVarCard* is *lbVarCard* which is at most as big as *minVarCard*. So if this estimation is at least as big as the lower bound of every association then lower bounds conformance is ensured.

Theorem 4.2.5 (Enhanced verification technique for lower bounds conformance)
Let r be an applicable rule set with target metamodel $mm = r_0|_{mv}|_{mm}$. It holds $lbConform(r)$, if

$\forall r_i \in r : lbConform(r_i)$

❑

Proof:

$$
\begin{array}{ll}
 & \forall r_i \in r : lbConform(r_i) \\
\overset{\text{Thm. 4.2.4}}{\Rightarrow} & \forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}, r_i \in r : lbConform(apply(m, r_i, (\emptyset, \emptyset)), r_i|_{mv_0}|_{mm}) \\
\overset{\text{Lemma 3.4.4}}{\Rightarrow} & \forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : lbConform(\bigcup_{r_i \in r}{}_{\mathfrak{m}}\, apply(m, r_i, (\emptyset, \emptyset))) \\
\overset{\text{Thm. 3.4.10}}{\Rightarrow} & \forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : lbConform(transform(m, r)) \\
\overset{\text{Def. 4.2.21}}{\Rightarrow} & lbConform(r)
\end{array}
$$

❏

To prove the lower bounds conformance of a rule set one can use Lemma 4.2.8 or Theorem 4.2.4 for the verification of the lower bounds conformance of the single rules.

4.2.4 Verification technique for metamodel conformance

Theorem 4.2.6 (Rule Set generates a valid Model)
Let r be a rule set. r generates a valid model, i.e. $transform(m, r)$ is a model transformation, if

(i) $ubConform(r) \wedge$

(ii) $lbConform(r)$

❏

Note: $lbConform(r)$ and $ubConform(r)$ imply that r is applicable.

Proof Sketch: According to Lemma 4.2.1 a model fragment's cardinalities have to comply to the multiplicities of the regarded metamodel to ensure that a newly generated model fragment is a model.

Theorem 4.2.6 is a direct consequence of the Definition of $lbConform(r)$ (cf. Def. 4.2.5) and the Definition of $ubConform(r)$ (cf. Def. 4.2.4). ❏

4.3 Glimpse towards Bijectivity

The third important property of rules and rule sets is bijectivity. In this section we provide some basic considerations towards bijectivity to present the current state of our ongoing work. We think that the Definitions provided here build a reasonable basis for further explorations.

Within some applications it doesn't suffice to do just a translation of one model to another model. Often it is necessary to do a re-translation which is e.g. needed for an enhanced CASE tool link. In the example of the link between the UML Suite and ASCET-SD, done in the AUTOMOTIVE project, components of the developed system can be added in both tools. So information about these components has to be translated and re-translated between both tools. Informally spoken what is needed is a notion of bijectivity which allows to do model transformations between two metamodels so that the source model being merged with the result of the inverse transformation isn't changed (more than necessary) even if the cycle of transformation and inverse transformation is done more than once. The inverse transformation has to construct the complete source fragment of the source model.

Definition 4.3.1 (Isomorphism ($m_0 \sim m_1$))
Two models m_0 and m_1 are called *isomorph* (short: $m_0 \sim m_1$), if it holds that:

(i) $m_0|_{mm} = m_1|_{mm}$

(ii) there exists a bijective mapping $map_o : m_0|_{OB_{mm_0}} \rightarrow m_1|_{OB_{mm_1}}$ with

$$\forall o_0 \in m_0|_{OB_{mm_0}} : map_o(o_0) = o_1 \wedge o_0|_{ot} = o_1|_{ot} \wedge o_0|_V = o_1|_V$$

(iii) there exists a bijective mapping $map_a : m_0|_{OA} \rightarrow m_1|_{OA}$ with

$$map_a(a_0) = a_1, \text{ with } \forall a_0, a_1 : a_0 = (o_0, o_1, oat, card) \wedge a_1 = (map_o(o_0), map_o(o_1), oat, card)$$

❍

This means in two isomorph models only the identifiers of objects may differ. The "structure" and the attribute values are identical.

Definition 4.3.2 (Inverse Rule (r_i^{-1}))
Let $r_i = (mv_0, mv_1)$ be a rule. Then the *inverse rule* of r is:

$$r_i^{-1} = (mv_1, mv_0)$$

❍

Definition 4.3.3 (Inverse Rule Set (r^{-1}))
Let r be a rule set consisting of the rules r_i. Then r^{-1} is the *inverse rule set* that consists of the inverse rules r_i^{-1}. ❍

Definition 4.3.4 ((Strict) Bijectivity)
A rule set r is *strictly bijective*, if and only if:

$$\forall m \in \mathbb{M}_{r_0|_{mv_0}|_{mm}} : transform(transform(m,r),r^{-1}) = m$$

A rule set r is *bijective* if and only if:

$$\forall m \in \mathbb{M}_{r_0|_{mv_0}|_{mm}} : m \sim transform(transform(m,r),r^{-1})$$

❍

Definition 4.3.5 (*sourceFrag*(m,r_i))
Let r be an arbitrary rule set with the source metamodel mm. Let $mfm \in MFM(m, r_i|_{mv_0})$ be a model fragment match sequence for an arbitrary model $m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$. Then the source fragment *sourceFrag* is:

$$sourceFrag(m,r_i) \mapsto mf$$
$$sourceFrag(m,r_i) = \bigcup_{\substack{\mathfrak{m} \\ j=0,\ldots,|mfm|-1}} mfm_j|_{mf}$$

❍

Definition 4.3.6 (Strictly bijecitve Rules)
A rule r_i is *strictly bijective*, if and only if

$$apply(apply(sourceFrag(m,r_i),r_i),r_i^{-1}) = sourceFrag(m,r_i) \text{ for } m \in \mathbb{M}_{mm} \text{ arbitrary}$$

❍

Definition 4.3.7 (Bijective Rules)
A rule r_i is *bijective*, if and only if

$$apply(apply(sourceFrag(m,r_i),r_i),r_i^{-1}) \sim sourceFrag(m,r_i) \text{ for } m \in \mathbb{M}_{mm} \text{ arbitrary}$$

❍

Definition 4.3.8 (Bijectivity)
Let $MV_1 = r_i|_{mv_1}$ be the sequence of all model variables of the rules' right hand sides. In order to allow a rule set to be *bijective* we demand that it holds for all model fragments mf, which have been generated by a "*merge*" of arbitrary instances (model fragments) of elements of MV_1: Every model fragment mf' which is structural congruent to a model variable mv_i originates (among others) also from the "*merge*" of an instance of mv_i and mf.

For such rule sets that are applicable in both directions it then holds that every rule application r_j has exactly one inverted rule application of the inverse rule r_j^{-1}. The successive application of

a rule and its inverse rule yields to a model fragment that differs from the source fragment only in the detail that attribute values that have the value $\diamondsuit$ in $r_j|_{mv_0}$ now also have this value in the newly created model fragment.

Such a rule set is called *partial bijective*. If it can be ensured that r and r^{-1} do capture all objects and attribute values in arbitrary source models each time ("*match*"), then the rule set is called *bijective*. ❍

Theorem 4.3.1 (Bijectivity)
A rule set r is strictly bijective, if

(i) all of its rules r_i are strictly bijective, and

(ii) it holds for all rules r_i, r_j in r:

$$r_i|_{mv_1}|_{OVA}|_{OVAT} \cap r_j|_{mv_1}|_{OVA}|_{OVAT} \neq \emptyset$$
$$\Rightarrow r_i = r_j$$

I.e. every association type occurs at maximal one rule's right hand side, and

(iii) no rule's right hand side can consist of only one object variable, if there is another rule's right hand side containing a model variable of the same type.

❑

Proof Sketch: Because of (ii) there exists exactly one rule for every association in a model m_1 generated by r, which has created this association. Therefore and because of (iii) only the rule that has created the corresponding model fragment does match in m_1 during the inverse transformation.

Trivially all fragments that have been created from a rule r_i do match again at the reverse transformation. Because of (i) this means that every inverse rule r_i^{-1} does match exactly the fragments that were created from the original rule r_i (and no others).

Since every rule is strictly bijective the merge of the model fragments generated by the inverted rule r_i^{-1} does also yield to $sourceFrag(m, r_i)$. Thus the merge of the model fragments that were generated by the application of the inverse rule r_i^{-1} yields to "$sourceFrag(m, \bigcup_i r_i)$" again. ❑

5 BOTL Extensions for Practical Use

The formalism and the properties presented in the previous sections from the base for specifying transformations between models based on metamodels. Up to now several important things are missing for the use of BOTL in practice with object oriented models:

Inheritance: Inheritance is one of the key features of object oriented methods. The BOTL core defined in the previous sections does not deal with inheritance. But from a statical, data oriented point of view, inheritance is rather simple. So in Section 5.2 the extension of the BOTL core with a simple treatment of inheritance is discussed.

Aggregation/Composition: Marking an association as an aggregation or composition puts further constraints on the possible models. Aggregation mean that objects with cyclic aggregation relationships are not allowed. Composition further restricts this by prohibiting an object to be part of more than one composition relationship. Especially composition is widely used in practice. So it would be desirably that BOTL supports both. Especially the notion of metamodel conformance has to be adapted. This will lead to further non-trivial adaption of the properties and the verification techniques defined in Section 4. This will be the subject of further research.

Mapping to practical used models: Though BOTL already uses UML-based notations no wide spread used notion of models is used for BOTL. Instead a simple formalization of the notion model is given. In practice models are defined in terms of Java, C++, E/R or UML. So a mapping between these practical used models and the BOTL models has to be specified. In Section 5.3 we define an exemplary mapping of BOTL metamodels to UML class models. Further mappings could be specified in a similar way.

In the following we describe first the notion of a BOTL metamodel in terms of a BOTL metamodel. This is necessary as afterwards this metamodel is used to define the notion of inheritance and a mapping of BOTL models to UML models.

5.1 BOTL Metamodel

In Section 3.1 a formal foundation for the BOTL metamodel was given. In Section 3.3 a UML profile for specifying BOTL metamodels was introduced. As we now use BOTL to describe some extensions we have to describe the BOTL metamodel in terms of BOTL.

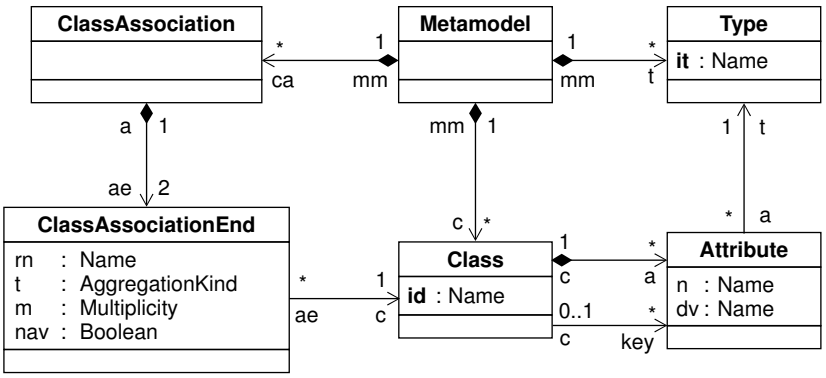

Figure 5.1: The BOTL metamodel mm_{core}

Figure 5.1 shows this metamodel represented using the given UML profile. The metamodel was derived systematically from the formalism. However there still remain some constraints that cannot be expressed in terms of a UML class diagram. These could be formalized with OCL constraints, but since we already provided a formal definition of the BOTL model we will only mention these constraints informally here.

Within the Figure 5.1 we use compositions. Thereby compositions can be replaced by simple associations as they have no meaning within BOTL. We use them anyway in the metamodel representation as e.g. a metamodel consists of a set of classes (cf. Def. 3.1.5 and Def. 3.1.8). That means a class could only be part of a metamodel once. So the composition with its informal meaning defined in the UML standard fits very well to express this fact.

As one can see the BOTL metamodel consists of a set of types, classes, and associations as formally defined in Section 3.1. The metamodel class itself is a singleton, i.e. there can only exist one instance of it. Every association between one or two classes consists of two association ends. According to the formalism a class association end has a role name, an aggregation type, a multiplicity, and a boolean value that indicates whether the association can be navigated in the given direction or not. Please note that the basic types of the associations in this class diagram all origin from the UML metamodel [OMG02].

Symmetric associations, i.e. those that have only one end in terms of the formalism, are represented by a ClassAssociation that has two identical ClassAssociationEnds. This representation differs a bit from our intuition regarding Definition 3.1.6 because identical association ends wouldn't be possible unless we use multi sets of association ends for the representation of class associations. However, we use this simplified notation for a more convenient handling of the model since a deterministic conversion between the two variants is obviously possible.

A class has a unique name id of the type Name as indicated by the tagged value {isPK = True}. Further every Class contains a set of attributes that have a name and refer to one of the metamodel's types. As documented in the BOTL formalism there is an additional constraint that determines that any two attribute instances that belong to the same class instance must have dif-

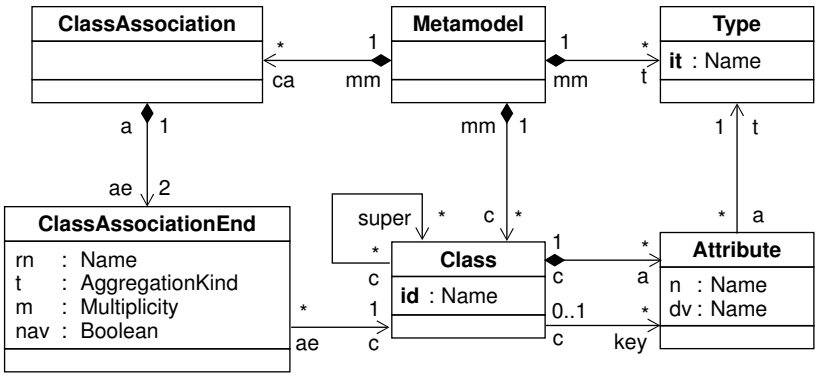

Figure 5.2: The BOTL metamodel with inheritance mm_{inh}

ferent names n. The key association points to those attributes that are necessary to determine the identity of the instances of a given class, according to the formalism.

5.2 Inheritance

Inheritance plays an important role within object orientation. With inheritance properties like attributes or methods of classes could be "inherited" to subclasses. As BOTL only deals with attributes, inheritance can be simply replaced by more verbose definitions.

Within BOTL inheritance is only used as a shortcut for attribute definitions. In rules inheritance is not regarded. Types within rules always match to exact the same types. Especially subclasses of a type never match to their super classes. Although it seems to be possible to extend rules with an appropriate "macro" mechanism this is not done yet, because one has to take care that the commutativity and associativity of rules is preserved and specialized rules for some subclasses are possible.

The goal of this section is to define a mapping of metamodels containing inheritance to BOTL core metamodels. We use the identity to map instance models that conform to metamodels with inheritance to those without inheritance.

5.2.1 BOTL Metamodel with Inheritance

We will use BOTL to define the mapping of metamodels with inheritance to BOTL metamodels. In Section 5.1 the BOTL metamodel was described in its own terms. Figure 5.2 shows an extended version now capable of a simple form of inheritance. A class association connecting a Class with itself has been added. Every Class could have a reference to an arbitrary number of super classes named super. A Class could be referenced by an also arbitrary number of subclasses.

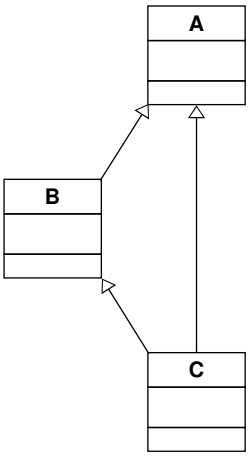

Figure 5.3: Multiple Inheritance of A

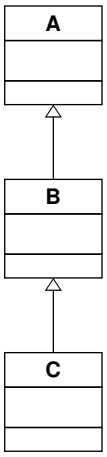

Figure 5.4: Single Inheritance

As usual rule-based formalisms are not capable of calculating a transitive closure. Therefore the super association shall contain the set of all super classes of a class. So for example the special case shown in Figure 5.3 cannot be distinguished from that one shown in Figure 5.4.

5.2.2 Transformation Rules

The rule set consisting of the rules r_0 to r_{10} (cf. Fig. 5.5 – 5.15) transforms models of the extended BOTL metamodel with inheritance into models of the core BOTL metamodel. As the Metamodel object is a singleton in both models we omit the transformation of this object and its associations. The rules r_0 to r_4 transform every model element except the super association into an equal model element of the core BOTL metamodel. The resulting core model is enriched by several new model elements which eliminate the inheritance relation.

r_0 (s. Fig. 5.5) transforms Types. r_1 (s. Fig. 5.6) does the same with Classes. r_2 and r_3 (s. Fig. 5.7 and 5.8) transform Attributes and their relationships. Thereby r_3 is needed to transform the key relationship modeling primary keys of a class. Note that the attribute values of Attribute are set to $\diamondsuit$ so that there is no conflict to r_2 detected (s. Sec. 5.2.3 below). r_4 (s. Fig. 5.9) transforms ClassAssociations and ClassAssoctiationEnds. All five rules maintain the identifiers of the core BOTL model besides the identifiers for ClassAssoctiationEnds which are not translated.

The rules r_5 to r_{10} translate models based on inheritance into models without inheritance. Therefore the super relationship is considered. For an easier understanding a small example in a shadowed box for a transformation between a model with and without inheritance is shown above the transformation arrow of each rule.

r_5 (s. Fig. 5.10) shows the meaning of the inheritance of an attribute. All attributes of a super class of a class become attributes of this class in the “core” model. Similarly to rule r_3 the primary key relationship is treated in an additional rule r_6 (s. Fig. 5.11). The rule r_7 to r_{10} deal

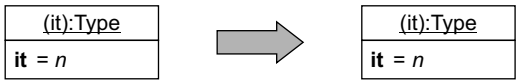

Figure 5.5: r_0

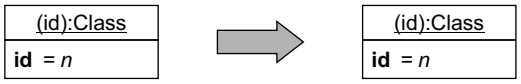

Figure 5.6: r_1

with different kinds of associations and their transformation. Note that the shown associations themselves are transformed by r_4 and so they are not transformed by r_5 to r_{10} again although that would not harm. In r_7 (s. Fig. 5.12) the case of an association between a super class A and a class C is shown. This leads to a new association between the subclass B and C. Rule r_8 (s. Fig. 5.13) shows the case of an association between a super class and its subclass. A new reflexive association on the subclass is added. Rule r_9 (s. Fig. 5.14) and r_{10} (s. Fig. 5.15) deal with the case of a reflexive association on the superclass of a class. In this case a total of three new associations have to be created.

As already mentioned with the given rule set the transformation of models conform to the enhanced BOTL metamodel into models conform to the core BOTL metamodel is specified. Now this rule set has to be examined for applicability and metamodel conformance which is done in the next two sections.

5.2.3 Applicability

Applicability of a rule set means that every transformation of a source model leads to a result different to $\perp$. That means that the rules produce a resulting model fragment for every possible source model.

According to Theorem 4.1.3 we have to ensure

(i) the applicability of every rule and

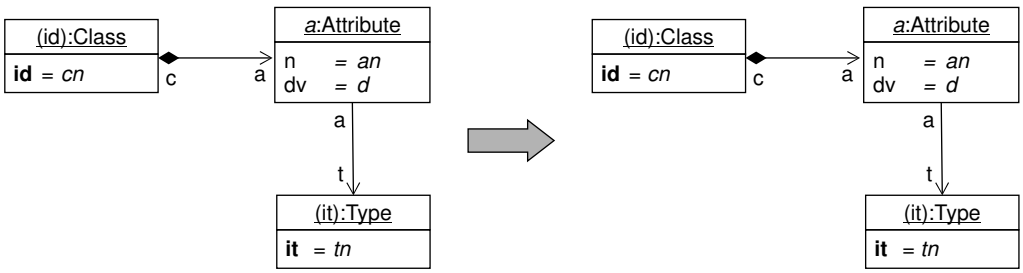

Figure 5.7: r_2

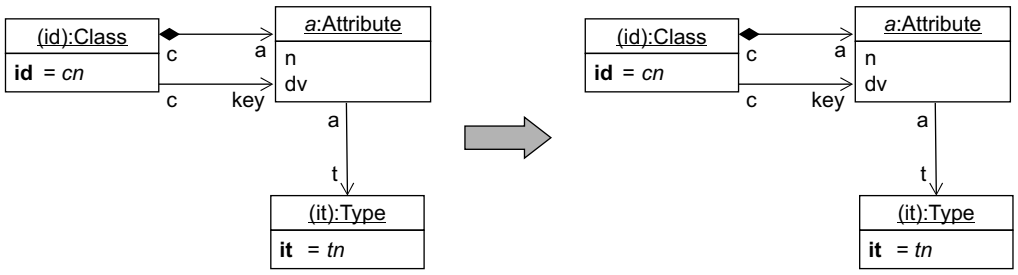

Figure 5.8: r_3

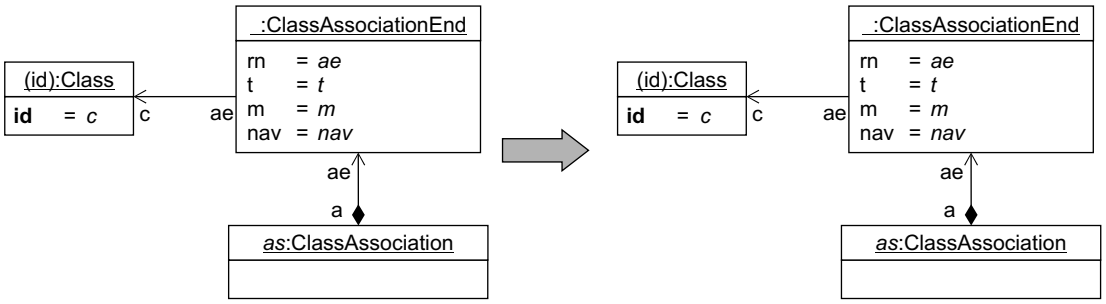

Figure 5.9: r_4

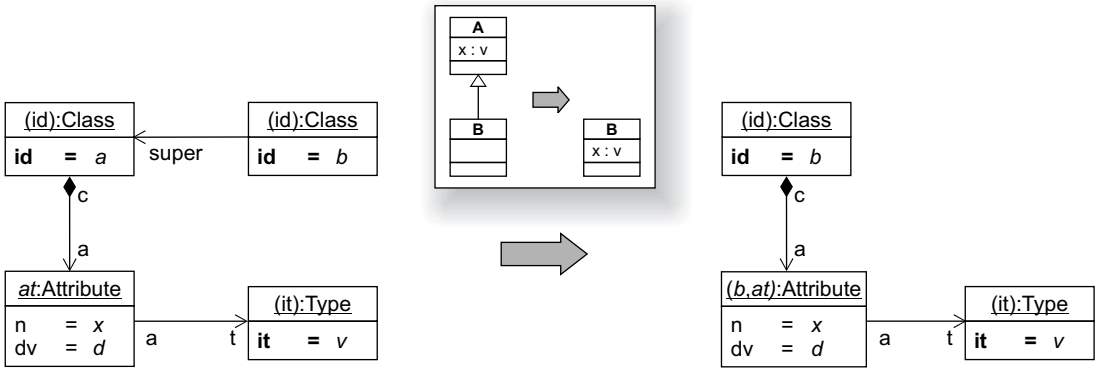

Figure 5.10: r_5: “attribute inheritance”

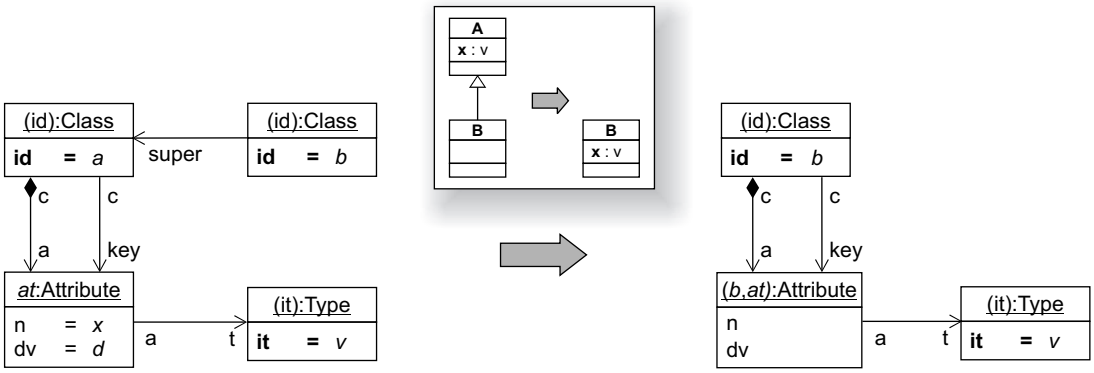

Figure 5.11: r_6: "attribute inheritance"

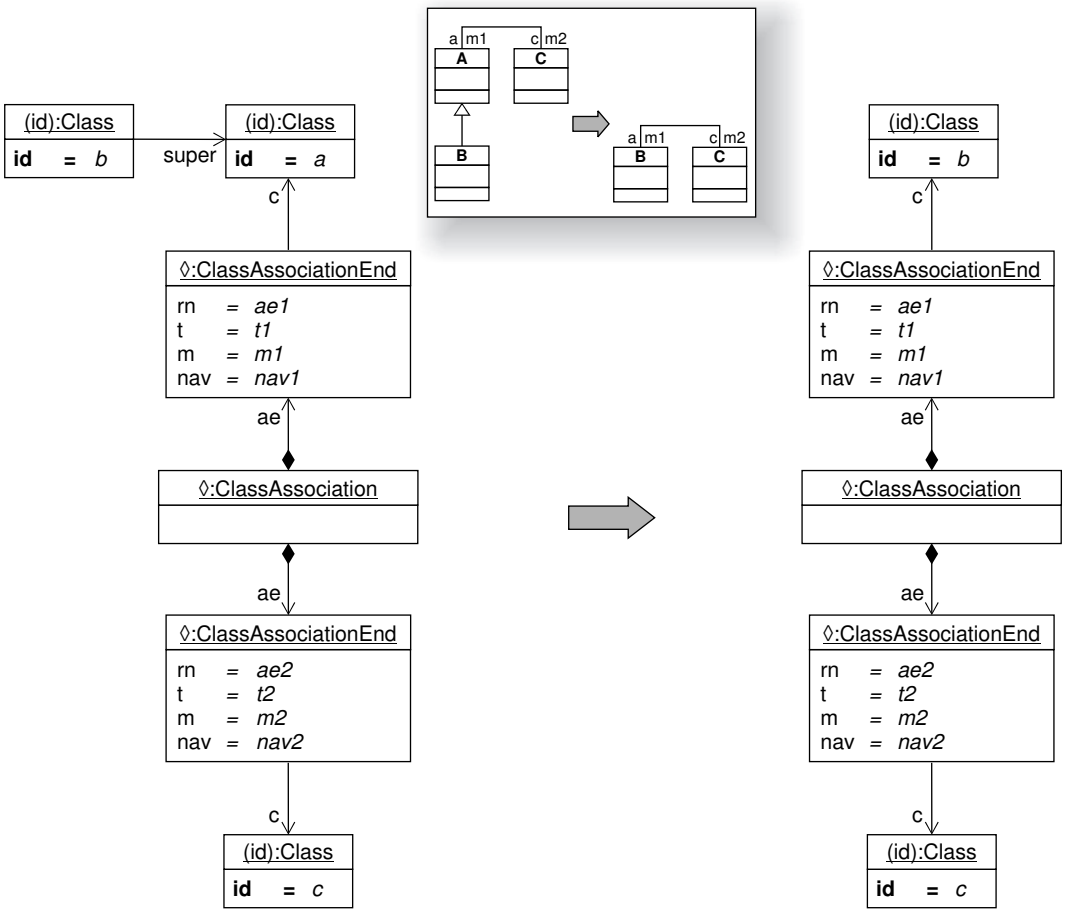

Figure 5.12: r_7: "inheritance of an association"

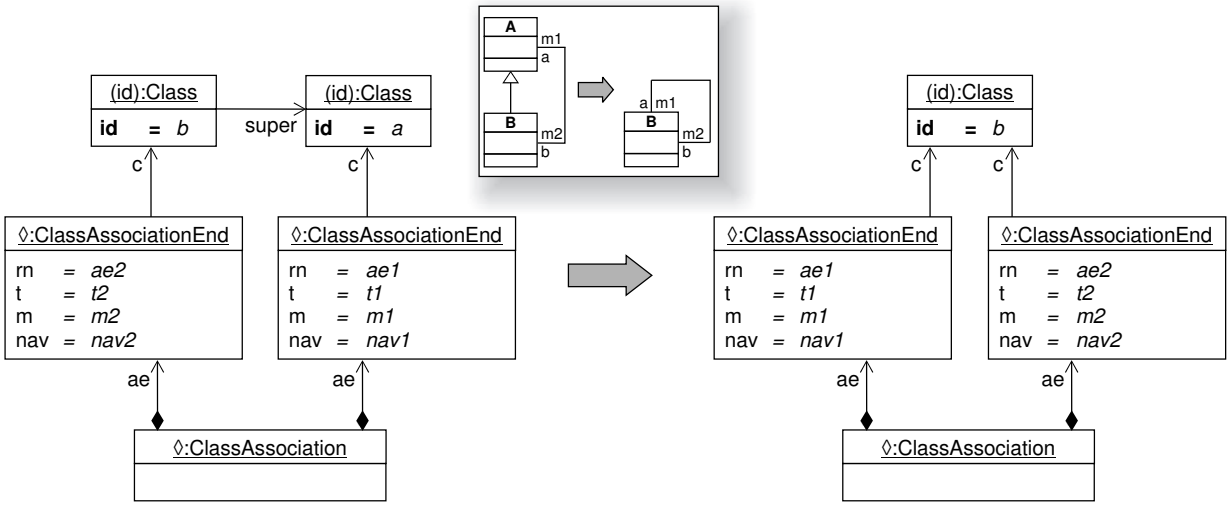

Figure 5.13: r_8: “inheritance of an association”

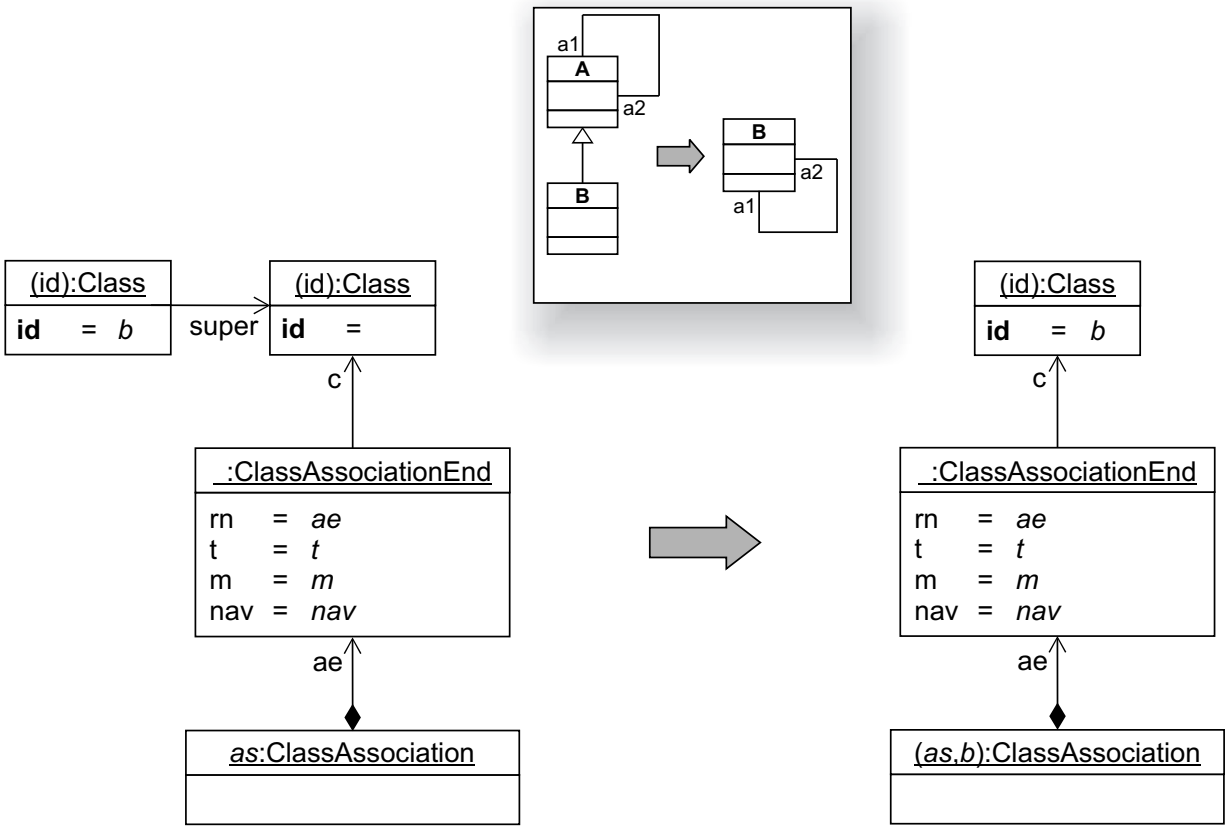

Figure 5.14: r_9: “inheritance of an association”

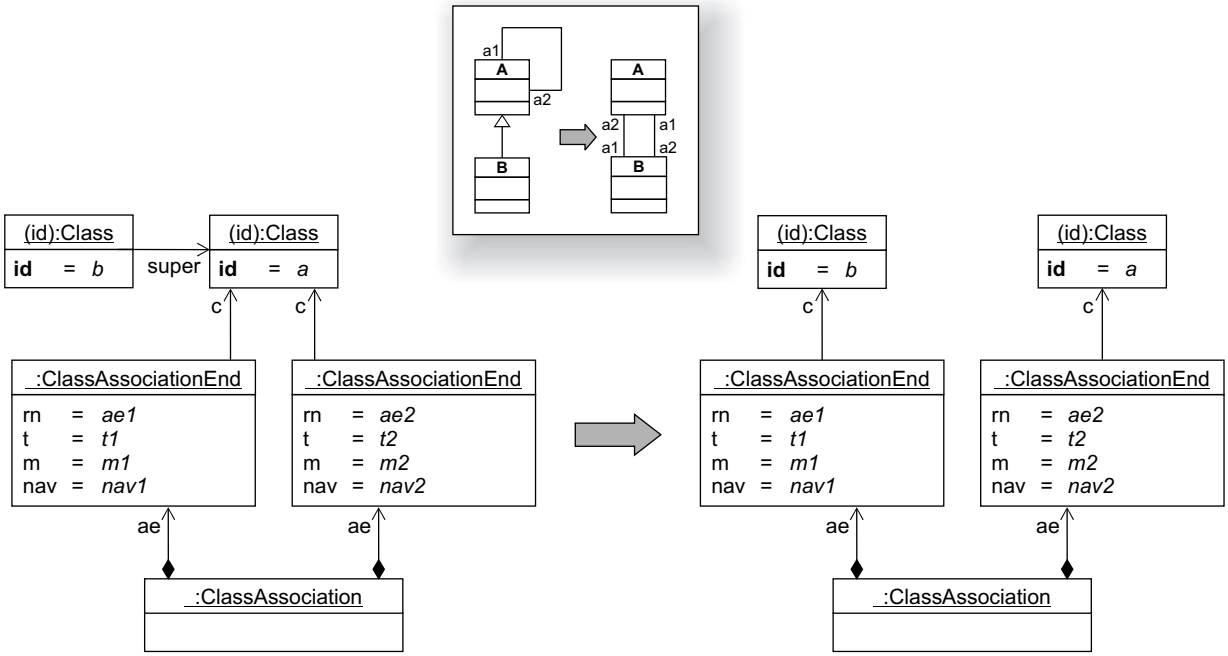

Figure 5.15: r_{10}: "inheritance of an association"

(ii) that there are no two right hand sides which are potentially contradictory.

So we first prove that every rule is applicable. From Theorem 4.1.2 we have to ensure that

(i) the equation system is unambiguously resolvable,

(ii) for every object variable on the right hand side at least one of the following statements hold:

- the object variable depends on object variables of the left hand side which determine the object variables of the left hand side on which all attributes are dependent,
- the identity is $\diamondsuit$, or
- all attributes are $\diamondsuit$,

and

(iii) for any two object variables of the right hand side with the same type it has to be ensured that the generated objects are *mergeable*. In the simplest case at all identifier term are equal to $\diamondsuit$. So it is ensured by construction that those generated objects are *mergeable* as they have different identities.

Applicability of r_0:

Let tv_0 be the object variables with the type Type on the left hand side of r_0. Similarly let tv_1 be the object variable on the right hand side of r_0. According to Definition 3.4.5 we get the following equations:

$$\begin{aligned}
GL^0_{id}: &\quad (match^0_o(tv_0)|oi = \varepsilon \quad \text{not considered according to Def. 3.4.5})\\
GL^0_{v}: &\quad match^0_o(tv_0).it = n\\
GL^1_{id}: &\quad pK(\{(it, match^1_o(tv_1).it)\}) = match^1_o(tv_1)|oi\\
GL^1_{v}: &\quad n = match^1_o(tv_1).it
\end{aligned}$$

Thus we can resolve the equational system and get an unambiguous solution for the right object variables' attributes and its identifiers:

$$\begin{aligned}
&match^1_o(tv_1)|oi = pK(\{(it, match^0_o(tv_0).it)\})\\
&match^1_o(tv_1).it = match^0_o(tv).it
\end{aligned}$$

So (i) holds.

For key attributes it always holds trivially: $dependsOn(ov, r_i) \overset{r_i|_{mv_0}}{\rightsquigarrow} attDependsOn(ov, a, r_i)$ Therefore (ii) holds.

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_0 is applicable.

Applicability of r_1:

Let cv_0 be the object variable with the types Class on the left hand side of r_1. Similarly let cv_1 be the object variable on the right hand side of r_1.

We get the following unambiguous solution of the equation system:

$$\begin{aligned}
&match^1_o(cv_1)|oi = pK(\{(id, match^0_o(cv_0).id)\})\\
&match^1_o(cv_1).id = match^0_o(cv).id
\end{aligned}$$

So (i) holds.

For key attributes it always holds trivially: $dependsOn(ov, r_i) \overset{r_i|_{mv_0}}{\rightsquigarrow} attDependsOn(ov, a, r_i)$ Therefore (ii) holds.

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_1 is applicable.

Applicability of r_2:

Let cv_0, av_0, and tv_0 be the object variables with the types Class, Attribute, and Type on the left hand side of r_2. Similarly let cv_1, av_1, and tv_1 be the object variables on the right hand side of r_2.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(cv_1)|oi &= pK(\{(id, match^0_o(cv_0).id)\})\\
match^1_o(cv_1).id &= match^0_o(cv_0).id\\
match^1_o(av_1)|oi &= match^0_o(av_0)|oi\\
match^1_o(av_1).n &= match^0_o(av_0).n\\
match^1_o(av_1).dv &= match^0_o(av_0).dv\\
match^1_o(tv_1)|oi &= pK(\{(it, match^0_o(tv_0).it)\})
\end{aligned}
$$

So (i) holds.

For key attributes it always holds trivially: $dependsOn(ov, r_i) \overset{r_i|_{mv_0}}{\rightsquigarrow} attDependsOn(ov, a, r_i)$ In the case of av_1 *attDependsOn* contains only av_0 for both attributes n and dv. $\{cv_0, av_0\} \overset{r_2|_{mv_0}}{\rightsquigarrow} \{av_0\}$ surely holds. Therefore (ii) holds.

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_2 is applicable.

Applicability of r_3:

Let cv_0, av_0, and tv_0 be the object variables with the types Class, Attribute, and Type on the left hand side of r_3. Similarly let cv_1, av_1, and tv_1 be the object variables on the right hand side of r_3.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(cv_1)|oi &= pK(\{(id, match^0_o(cv_0).id)\})\\
match^1_o(cv_1).id &= match^0_o(cv_0).id\\
match^1_o(av_1)|oi &= match^0_o(av_0)|oi\\
match^1_o(av_1).n &= \diamondsuit\\
match^1_o(av_1).dv &= \diamondsuit\\
match^1_o(tv_1)|oi &= pK(\{(it, match^0_o(tv_0).it)\})
\end{aligned}
$$

So (i) holds.

For key attributes it always holds trivially: $dependsOn(ov, r_i) \overset{r_i|_{mv_0}}{\rightsquigarrow} attDependsOn(ov, a, r_i)$ Therefore (ii) holds.

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_3 is applicable.

Applicability of r_4:

Let cv_0, $caev_0$, and cav_0 be the object variables with the types Class, ClassAssociationEnd, and ClassAssociation on the left hand side of r_4. Similarly let cv_1, $caev_1$, and cav_1 be the object variables on the right hand side of r_4.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(cv_1)|oi &= pK(\{(id, match^0_o(cv_0).id)\}) \\
match^1_o(cv_1).id &= match^0_o(cv_0).id \\
match^1_o(caev_1)|oi &= genID(\ldots) \\
match^1_o(caev_1).rn &= match^0_o(caev_0).rn \\
match^1_o(caev_1).t &= match^0_o(caev_0).t \\
match^1_o(caev_1).m &= match^0_o(caev_0).m \\
match^1_o(caev_1).nav &= match^0_o(caev_0).nav \\
match^1_o(cav_1)|oi &= match^0_o(cav_0)|oi
\end{aligned}
$$

So (i) holds.

$dependsOn(caev_1)$ yields to $\diamondsuit$. Therefore (ii) holds. *genID* generates an unique identifier.

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_4 is applicable.

Applicability of r_5:

Let cv_0, $csubv_0$, av_0, and tv_0 be the object variables with the types Class (super class), Class (subclass), Attribute, and Type on the left hand side of r_5. Thereby cv_0 is the super class with the id Term a and $csubv_0$ is the subclass with the id Term b. Similarly let $csubv_1$, av_1, and tv_1 be the object variables on the right hand side of r_5.

We get the following unambiguous solution of the equation system:

$match^1_o(csubv_1)|oi = pK(\{(id, match^0_o(csubv_0).id)\})$

$$
\begin{aligned}
match^1_o(csubv_1).id &= match^0_o(csubv_0).id \\
match^1_o(av_1)|oi &= uK(match^0_o(csubv_0).id, match^0_o(av_0).n) \\
match^1_o(av_1).n &= match^0_o(av_0).n \\
match^1_o(av_1).dv &= match^0_o(av_0).dv \\
match^1_o(tv_1)|oi &= pK(\{(it, match^0_o(tv_0).it)\}) \\
match^1_o(tv_1).it &= match^0_o(tv_0).it
\end{aligned}
$$

So (i) holds.

Again all attributes of every class depend only on one source class which is also the source for calculating the regarding identity. In the case of av_1 it holds:

$$dependsOn(av_1) = \{csubv_0, av_0\} \overset{r_5|_{mv_0}}{\rightsquigarrow} attDependsOn(av_1) = \{av_0\}$$

Therefore (ii) holds. Note that the tuple (b, x) generates a new key which can be only generated by another 2-tuple (cf. r_6).

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_5 is applicable.

Applicability of r_6:

Let cv_0, $csubv_0$, av_0, and tv_0 be the object variables with the types Class (super class), Class (subclass), Attribute, and Type on the left hand side of r_6. Thereby cv_0 is the super class with the id Term a and $csubv_0$ is the subclass with the id Term b. Similarly let $csubv_1$, av_1, and tv_1 be the object variables on the right hand side of r_6.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(csubv_1)|oi &= pK(\{(id, match^0_o(csubv_0).id)\}) \\
match^1_o(csubv_1).id &= match^0_o(csubv_0).id \\
match^1_o(av_1)|oi &= uK(match^0_o(csubv_0).id, match^0_o(av_0).n) \\
match^1_o(av_1).n &= \diamondsuit \\
match^1_o(av_1).dv &= \diamondsuit \\
match^1_o(tv_1)|oi &= pK(\{(it, match^0_o(tv_0).it)\}) \\
match^1_o(tv_1).it &= match^0_o(tv_0).it
\end{aligned}
$$

So (i) holds.

No attribute values (except key attributes) are set. So (ii) holds. Note that the tuple (b, x) generates a new key which can be only generated by another 2-tuple (cf. r_5).

(iii) holds as there are no object variables with the same type on the right hand side.

Therefore r_6 is applicable.

Applicability of r_7:

Let acv_0, bcv_0, ccv_0, $cae1v_0$, $cae2v_0$, and cav_0 be the object variables with the types Class and the id term *a*, Class and the id term *b*, Class and the id term *c*, ClassAssociationEnd and the rn term *ae1*, ClassAssociationEnd and the rn term *ae2*, and ClassAssociation on the left hand side of r_7. Similarly let bcv_1, ccv_1, $cae1v_1$, $cae2v_1$, and cav_1 be the object variables on the right hand side of r_7.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(bcv_1)|oi &= pK(\{(id, match^0_o(bcv_0).id)\})\\
match^1_o(bcv_1).id &= match^0_o(bcv_0).id\\
match^1_o(cae1v_1)|oi &= genID(\dots)\\
match^1_o(cae1v_1).rn &= match^0_o(cae1v_0).rn\\
match^1_o(cae1v_1).t &= match^0_o(cae1v_0).t\\
match^1_o(cae1v_1).m &= match^0_o(cae1v_0).m\\
match^1_o(cae1v_1).nav &= match^0_o(cae1v_0).nav\\
match^1_o(cav_1)|oi &= genID(\dots)\\
match^1_o(cae2v_1)|oi &= genID(\dots)\\
match^1_o(cae2v_1).rn &= match^0_o(cae2v_0).rn\\
match^1_o(cae2v_1).t &= match^0_o(cae2v_0).t\\
match^1_o(cae2v_1).m &= match^0_o(cae2v_0).m\\
match^1_o(cae2v_1).nav &= match^0_o(cae2v_0).nav\\
match^1_o(ccv_1)|oi &= pK(\{(id, match^0_o(ccv_0).id)\})\\
match^1_o(ccv_1).id &= match^0_o(ccv_0).id
\end{aligned}
$$

So (i) holds.

Only with the two object variables with the type ClassAssociationEnd attributes are set. Both variables have the identifier set to $\diamondsuit$. So (ii) holds. Note that these $\diamondsuit$ values are replaced by a generated identifier which is unique. Example (with correct *genID* term):

$$match^1_o(cae1v_1)|oi = genID(7, v, genNum(cae1v_1, r_7|_{mv_1}|_{OVB}))$$

Thereby v is the actual number of the model fragment relation "application" ranging from 0 to the number of the concrete model fragment match sequence $|mfm| - 1$.

(iii) holds as both object variables with the type ClassAssociationEnd on the right hand side have the identifier set to $\diamondsuit$.

Therefore r_7 is applicable.

Applicability of r_8:

Let acv_0, bcv_0, $cae1v_0$, $cae2v_0$, and cav_0 be the object variables with the types Class and the id term *a*, Class and the id term *b*, ClassAssociationEnd and the rn term *ae1*, ClassAssociationEnd and the rn term *ae2*, and ClassAssociation on the left hand side of r_8. Similarly let bcv_1, $cae1v_1$, $cae2v_1$, and cav_1 be the object variables on the right hand side of r_8.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(bcv_1)|oi &= pK(\{(id, match^0_o(bcv_0).id)\}) \\
match^1_o(bcv_1).id &= match^0_o(bcv_0).id \\
match^1_o(cae1v_1)|oi &= genID(\dots) \\
match^1_o(cae1v_1).rn &= match^0_o(cae1v_0).rn \\
match^1_o(cae1v_1).t &= match^0_o(cae1v_0).t \\
match^1_o(cae1v_1).m &= match^0_o(cae1v_0).m \\
match^1_o(cae1v_1).nav &= match^0_o(cae1v_0).nav \\
match^1_o(cav_1)|oi &= genID(\dots) \\
match^1_o(cae2v_1)|oi &= genID(\dots) \\
match^1_o(cae2v_1).rn &= match^0_o(cae2v_0).rn \\
match^1_o(cae2v_1).t &= match^0_o(cae2v_0).t \\
match^1_o(cae2v_1).m &= match^0_o(cae2v_0).m \\
match^1_o(cae2v_1).nav &= match^0_o(cae2v_0).nav
\end{aligned}
$$

So (i) holds.

Only with the two object variables with the type ClassAssociationEnd attributes are set. Both variables have the identifier set to $\diamondsuit$. So similarly to r_7 (ii) holds.

(iii) holds as both object variables with the type ClassAssociationEnd on the right hand side have the identifier set to $\diamondsuit$.

Therefore r_8 is applicable.

Applicability of r_9:

Let acv_0, bcv_0, $caev_0$, and cav_0 be the object variables with the types Class and the id term *a*, Class and the id term *b*, ClassAssociationEnd, and ClassAssociation on the left hand side of r_9. Similarly let bcv_1, $caev_1$, and cav_1 be the object variables on the right hand side of r_9.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(bcv_1)|oi &= pK(\{(id, match^0_o(bcv_0).id)\}) \\
match^1_o(bcv_1).id &= match^0_o(bcv_0).id \\
match^1_o(caev_1)|oi &= genID(\dots) \\
match^1_o(caev_1).rn &= match^0_o(caev_0).rn \\
match^1_o(caev_1).t &= match^0_o(caev_0).t \\
match^1_o(caev_1).m &= match^0_o(caev_0).m \\
match^1_o(caev_1).nav &= match^0_o(caev_0).nav \\
match^1_o(cav_1)|oi &= uK(match^0_o(cav_0).id, match^0_o(bcv_0).id)
\end{aligned}
$$

So (i) holds. Again (ii) and (iii) hold trivially.

Therefore r_9 is applicable.

Applicability of r_{10}:

Let acv_0, bcv_0, $cae1v_0$, $cae2v_0$, and cav_0 be the object variables with the types Class and the id term *a*, Class and the id term *b*, ClassAssociationEnd and the rn term *ae1*, ClassAssociationEnd and the rn term *ae2*, and ClassAssociation on the left hand side of r_{10}. Similarly let acv_1, bcv_1, $cae1v_1$, $cae2v_1$, and cav_1 be the object variables on the right hand side of r_{10}.

We get the following unambiguous solution of the equation system:

$$
\begin{aligned}
match^1_o(acv_1)|oi &= pK(\{(id, match^0_o(acv_0).id)\}) \\
match^1_o(acv_1).id &= match^0_o(acv_0).id \\
match^1_o(bcv_1)|oi &= pK(\{(id, match^0_o(bcv_0).id)\}) \\
match^1_o(bcv_1).id &= match^0_o(bcv_0).id \\
match^1_o(cae1v_1)|oi &= genID(\dots) \\
match^1_o(cae1v_1).rn &= match^0_o(cae1v_0).rn \\
match^1_o(cae1v_1).t &= match^0_o(cae1v_0).t \\
match^1_o(cae1v_1).m &= match^0_o(cae1v_0).m \\
match^1_o(cae1v_1).nav &= match^0_o(cae1v_0).nav
\end{aligned}
$$

$$match^1_o(cav_1)|oi = genID(\dots)$$
$$match^1_o(cae2v_1)|oi = genID(\dots)$$
$$match^1_o(cae2v_1).rn = match^0_o(cae2v_0).rn$$
$$match^1_o(cae2v_1).t = match^0_o(cae2v_0).t$$
$$match^1_o(cae2v_1).m = match^0_o(cae2v_0).m$$
$$match^1_o(cae2v_1).nav = match^0_o(cae2v_0).nav$$

So (i) holds. Again (ii) and (iii) holds trivially.

Therefore r_{10} is applicable.

Applicability of r:

We have shown above, that every rule r_i of the rule set r is applicable. Therefore (i) of Theorem 4.1.3 holds. (ii) of Theorem 4.1.3 requires, that there are no two object variables on the right hand side of two rules with the same type which possibly lead to contradictory attribute values. We don't have to look at object variables with the identifier term $\diamondsuit$ as these are surely unique. All attribute terms for attributes which are no key attributes have to be considered. We don't have to look at object variables with only $\diamondsuit$ terms for those attribute values or at object variables with only key attributes. Therefore the object variables of the type Class, Type, and ClassAssociation (only key attributes), and ClassAssociationEnd ($\diamondsuit$ identifier) can be ignored safely. Only object variables with the type Attribute have to be examined.

Within r_2, r_3, r_5, and r_6 object variables of the type Attribute appear on the right hand side. From the identifier terms it is obvious, that only r_2 with r_3, or r_5 with r_6 can clash. A 2-tuple (*(b,x)*) can never clash with a simple value (*a*). For both pairs of rules it holds, that only one rule set the attribute terms to values different to $\diamondsuit$.

Therefore also (ii) of Theorem 4.1.3 holds. This leads to the conclusion that r is applicable.

5.2.4 Metamodel Conformance

The proof of metamodel conformance is necessary to ensure that the given rule set produces only model fragments which are models conform to the target metamodel. Besides the applicability shown above metamodel conformance depends on the possible cardinalities of object associations which are generated in a rule set application.

Again verification techniques are used to decide if a rule set is metamodel conform. Theorem 4.2.6 says that a rule set generates a valid model if it is

(*i*) *lbConform*, that means the structure of the rules and the source metamodel ensure that all cardinalities of outgoing object associations of any object in the target model are at least as many as required by the multiplicities of the target metamodel, and

(*ii*) *ubConform*, that means the structure of the rules and the source metamodel ensure that all cardinalities of outgoing object associations of any object in the target model are at most as many as required by the multiplicities of the target metamodel.

Lower bounds conformance

The lower bounds conformance ($lbConform(r)$) of a rule set r can be proven by the application of one of the following verification techniques:

Simple verification technique (*varLbConform*): If every rule in a rule set is *varLbConform* then whole rule set is *lbConform* (cf. Theorem 4.2.3). The check if a rule is *varLbConform* is very simple. It simply has to be checked for every rule that every target object variable has at least as many outgoing object variable associations as the lower bounds of its classes associations specify.

Enhanced verification technique ($lbConform(r_i)$): If it holds that for every rule that every target object variable the estimation *lbCard* for every possible outgoing association is at least as big as the lower bound of that specified by its classes associations, then the whole rule set is *lbConform*.

The simple and the enhanced verification technique can be mixed arbitrary for different rules (Theorem 4.2.8, Theorem 4.2.4, and Theorem 4.2.5).

In the following we prove the lower bounds conformance of the rule $r_0 - r_{10}$. As already stated we neglect the singleton Metamodel. The according composition associations could be included easily.

Lower bounds conformance of r_0 The target model variable consists only of an object variable of type Class. From the metamodel depicted in Figure 5.1 we can easily see that all outgoing associations have a lower bound of 0 as the association to Metamodel is ignored. So it holds: $varLbConform(r_0|_{mv_1}, mm_{core})$

Lower bounds conformance of r_1 The target model variable consists only of an object variable of type Type. From the metamodel depicted in Figure 5.1 we can easily see that all outgoing associations have a lower bound of 0. So it holds: $varLbConform(r_1|_{mv_1}, mm_{core})$

Lower bounds conformance of r_2 Only the object variable of type Attribute has to be considered as this is the only one with outgoing associations greater than 0. In mm_{core} it is specified, that every Attribute is contained in one Class and refers to one Type. As both association are created together with an Attribute the lower bounds conformance is ensured. So it holds: $varLbConform(r_2|_{mv_1}, mm_{core})$

Lower bounds conformance of r_3 From the lower bounds point of view r_3 is very similar to r_2. So it holds: $varLbConform(r_3|_{mv_1}, mm_{core})$

Lower bounds conformance of r_4 In r_4 $varLbConform(r_4|_{mv_1}, mm_{core})$ doesn't hold as the object variable of the type ClassAssociation has only one outgoing association to ClassAssociationEnds. So the enhanced verification technique has to be applied. For the three participating classes only the two shown associations have to be considered as there are no other outgoing associations with an lower bound greater than 0 (despite the associations to Metamodel which are neglected).

Let cv_1, $caev_1$, and cav_1 be the object variables of the types Class, ClassAssociationEnd, and ClassAssociation on the right hand side of r_4. Further let ova_{aec} and ova_{aae} be the object variable associations between the object variables of the types ClassAssociationEnd and the Class, and ClassAssociation and ClassAssociationEnd on the right hand side. Let cv_0, $caev_0$, and cav_0 be the object variables of the types Class, ClassAssociationEnd, and ClassAssociation on the left hand side of r_4.

We have to prove the following equations:

- $lbCard(AE_{aec}, ae_{aec,c}, r_4) \geq 1$, for the association from the ClassAssociationEnd to the Class,
- $lbCard(AE_{aec}, ae_{aec,ae}, r_4) \geq 0$, for the association from the Class to the ClassAssociationEnd,
- $lbCard(AE_{aae}, ae_{aae,ae}, r_4) \geq 2$, for the association from the ClassAssociation to the ClassAssociationEnd, and
- $lbCard(AE_{aae}, ae_{aae,a}, r_4) \geq 1$, for the association from the ClassAssociationEnd to the ClassAssociation.

According to Definition 4.2.27 and Definition 4.2.26 we can calculate the following values:

$$
\begin{aligned}
lbCard(AE_{aec}, ae_{aec,c}, r_4) &= min_{ova:ova|_{OVAT=AE_{aec}}}(lbVarCard(ova, ae_{aec,c}, r_4)) \\
&= lbVarCard(ova_{aec}, ae_{aec,c}, r_4) \\
&\overset{\text{Tab. 4.2.26, } (xii)}{=} 1 \\
&\geq 1
\end{aligned}
$$

$$
\begin{aligned}
lbCard(AE_{aec}, ae_{aec,ae}, r_4) &= min_{ova:ova|_{OVAT=AE_{aec}}}(lbVarCard(ova, ae_{aec,ae}, r_4)) \\
&= lbVarCard(ova_{aec}, ae_{aec,ae}, r_4) \\
&\overset{\text{Tab. 4.2.26, } (xvi)}{=} minmatch(r_4|_{mv_0}|_{mm}, r_4|_{mv_0}, dependsOn(cv_1, r_4)) \\
&\quad \cdot ova_{aec}|_{cardv} \\
&= minmatch(r_4|_{mv_0}|_{mm}, r_4|_{mv_0}, \{cv_0\}) \cdot 1 \\
&= 0 \\
&\geq 0
\end{aligned}
$$

$$
\begin{aligned}
lbCard(AE_{aae}, ae_{aae,ae}, r_4) &= min_{ova:ova|_{OVAT=AE_{aae}}}(lbVarCard(ova, ae_{aae,ae}, r_4)) \\
&= lbVarCard(ova_{aae}, ae_{aae,ae}, r_4) \\
&\overset{\text{Tab. 4.2.26, } (xvi)}{=} minmatch(r_4|_{mv_0}|_{mm}, r_4|_{mv_0}, dependsOn(cav_1, r_4)) \\
&\quad \cdot ova_{aae}|_{cardv} \\
&= minmatch(r_4|_{mv_0}|_{mm}, r_4|_{mv_0}, \{cav_0\}) \cdot 1 \\
&= 2 \\
&\geq 2
\end{aligned}
$$

$$
\begin{aligned}
lbCard(AE_{aae}, ae_{aae,a}, r_4) &= min_{ova:ova|_{OVAT=AE_{aae}}}(lbVarCard(ova, ae_{aae,a}, r_4)) \\
&= lbVarCard(ova_{aae}, ae_{aae,a}, r_4) \\
&\overset{\text{Tab. 4.2.26, } (xii)}{=} 1 \\
&\geq 1
\end{aligned}
$$

Thus r_4 is *lbConform*.

Lower bounds conformance of r_5 Only the object variable of type Attribute has to be considered as this is the only one with outgoing associations greater than 0. In mm_{core} it is specified, that every Attribute is contained in one Class and refers to one Type. As both association are created together with an Attribute the lower bounds conformance is ensured. So it holds: $varLbConform(r_5|_{mv_1}, mm_{core})$

Lower bounds conformance of r_6 From the lower bounds point of view r_6 is very similar to r_5. So it holds: $varLbConform(r_6|_{mv_1}, mm_{core})$

Lower bounds conformance of r_7 Every ClassAssociationEnd has to refer to at least one Class and has to be part of at least one ClassAssociation. Every ClassAssociation has to contain at least two ClassAssociationEnds. All three conditions are ensured by the rule r_7 by construction. So it holds: $varLbConform(r_7|_{mv_1}, mm_{core})$

Lower bounds conformance of r_8 Similarly to r_7 it holds: $varLbConform(r_8|_{mv_1}, mm_{core})$

Lower bounds conformance of r_9 Similarly to r_4 it holds: $lbConform(r_9|_{mv_1}, mm_{core})$

Lower bounds conformance of r_{10} Similarly to r_7 and r_8 $varLbConform(r_{10}|_{mv_1}, mm_{core})$ holds.

Lower bounds conformance of r As for all rules r_i part of the rule set r are *lbConform* also it holds: $lbConform(r)$

Upper bounds conformance

Similar to the lower bounds conformance it suffices to apply a given verification technique for upper bounds conformance. But here the situation is slightly more complicate. First of all the estimation of the maximal possible number of association created by a given object variable association and a given rule has to be calculated. Afterwards those numbers for outgoing associations of the same type have to be added. But within this sum some redundant summands have to be neglected.

So first of all we detect summands which are redundant. Afterwards for the relevant outgoing object variable associations we have to calculate $ubVarCard(ova, ae, r_i)$.

Calculation of *redundant* $redundant(ova, r_i, r)$ holds if a given object variable associations of a given rule is redundant, i.e. if all instances created by this variable will be generated by another object variable associations of another rule.

redundant holds if there exists another rule r_j which left hand side is a substructure of r_i, i.e. $r_j \sqsubseteq r_i$. Afterwards the right hand side of the rules have to be compared. If there is an object variable association in r_j with the same type as that of *ova* and object variables at its ends which are generating the same objects as those of r_i, then *redundant* holds.

In Table 5.1 for every rule the rules which left hand sides substructures of the rule are shown. So for example in rule r_2 the left hand side consist of a object variables of type Class and Type both being the left hand sides of other rules r_1 and r_0. Note that we only compare the structure of model variables not the contents of identifier or attribute terms.

r_i	r_j with $i \neq j$ and $r_j\|_{mv_0} \sqsubseteq r_i\|_{mv_0}$
r_0	$\emptyset$
r_1	$\emptyset$
r_2	$\{r_0, r_1\}$
r_3	$\{r_0, r_1, r_2\}$
r_4	$\{r_1\}$
r_5	$\{r_0, r_1, r_2\}$
r_6	$\{r_0, r_1, r_2, r_3, r_5\}$
r_7	$\{r_1, r_4, r_9\}$
r_8	$\{r_1, r_4, r_9\}$
r_9	$\{r_1, r_4\}$
r_{10}	$\{r_1, r_4, r_9\}$

Table 5.1: Rules and substructures of their left hand side

Obviously we are not interested in rule which generate no object associations. So r_0 and r_1 are ignored in the following. For *redundant* it is also necessary to calculate $OVP(r_{super}, r_{sub})$ for every rule. This contains pairs of object variables which generate the "same" objects. The results of *OVP* can be seen in Table 5.2. Thereby we use the same names for object variables as those introduced in Section 5.2.3. Lines for which *OVP* is equal to $\emptyset$ are as well neglected as the rule r_0 and r_1. For the calculation of *OVP* the identifier terms have to be compared. Here it is helpful to know that $\diamondsuit$ is never equal to anything (not even to $\diamondsuit$), and that a term consisting of a 2-tuple is always different to a 1-tuple.

As the calculation of all necessary *ubVarCard* values is lengthy we will concentrate on some interesting examples.

Composition between Class and Attribute The first association analyzed will be the composition between a Class and an Attribute. First of all we notice that a Class consists of an arbitrary number of Attributes. So the outgoing association of Class objects to Attributes are surely upper bound conform.

But the other direction is of more interest. Every Attribute is part of at most one Class. So we have an upper bound *ub* of 1. We name the according object variable associations in the rules r_2, r_3, r_5, and r_6 $r_i.ova_{ca}$ with an appropriate i. The class association ends are ae_c and ae_a.

From Theorem 4.2.2 we know that we have to sum up all *ubVarCard* terms starting at object variables of type Attribute with similar identifiers.

r_{super}	r_{sub}	$OVP(r_{super}, r_{sub})$
r_3	r_2	$\{(r_3.cv_1, r_2.cv_1), (r_3.av_1, r_2.av_1), (r_3.tv_1, r_2.tv_1)\}$
r_5	r_2	$\{(r_5.csubv_1, r_2.cv_1), (r_5.tv_1, r_2.tv_1)\}$
r_6	r_2	$\{(r_6.csubv_1, r_2.cv_1), (r_6.tv_1, r_2.tv_1)\}$
r_6	r_3	$\{(r_6.csubv_1, r_3.cv_1), (r_6.tv_1, r_3.tv_1)\}$
r_6	r_5	$\{(r_6.csubv_1, r_5.cv_1), (r_6.av_1, r_5.av_1), (r_6.tv_1, r_5.tv_1)\}$
r_7	r_4	$\{(r_7.ccv_1, r_4.cv_1)\}$
r_7	r_9	$\{(r_7.bcv_1, r_9.bcv_1)\}$
r_8	r_4	$\{(r_8.bcv_1, r_4.cv_1)\}$
r_8	r_9	$\{(r_8.bcv_1, r_9.bcv_1)\}$
r_{10}	r_4	$\{(r_{10}.acv_1, r_4.cv_1)\}$
r_{10}	r_9	$\{(r_{10}.bcv_1, r_9.bcv_1)\}$

Table 5.2: The results of $OVP(r_{super}, r_{sub})$

From the rules we can state that only the following two pairs of object variable identifiers are similar as the first ones are 1-tuples and the second ones are 2-tuples:

$r_2.av_1|_{oiv} \sim r_3.av_1|_{oiv}$
$r_5.av_1|_{oiv} \sim r_6.av_1|_{oiv}$

From Table 5.1 we know that it holds:

$r_2|_{mv_0} \sqsubseteq r_3|_{mv_0}$
$r_5|_{mv_0} \sqsubseteq r_6|_{mv_0}$

Together with Table 5.2 and Definition 4.2.19 we know that it holds:

$redundant(r_3.ova_{ca}, r_3, r)$
$redundant(r_6.ova_{ca}, r_6, r)$

The left hand side of r_3 is a superset of the left hand side of r_2 and objects of type Class and Attribute generated by r_3 are also generated by r_2. Note that the applicability proven above ensures that those objects can be merged. So also all associations generated by $r_3.ova_{ca}$ are generated by $r_2.ova_{ca}$. So $r_3.ova_{ca}$ is redundant.

According to Theorem 4.2.2 we have to calculate the relevant summands. Exactly the following similar relations hold for object variables of type Attribute:

$r_2.av_1 \sim r_3.av_1$
$r_5.av_1 \sim r_6.av_1$

So from the Definition of *relevant* (Def. 4.2.20) we can conclude that we have to calculate two sums both consisting of exactly one summand. The summands consist of the outgoing associations from the object variable of the type Attribute in rules r_2 and r_5.

This means we have to calculate $ubVarCard(r_2.ova_{ca}, ae_c, r_2)$ and $ubVarCard(r_5.ova_{ca}, ae_c, r_5)$.

$dependsOn(r_2.av_1, r_2) = \{r_2.av_0\}$
$dependsOn(r_2.cv_1, r_2) = \{r_2.cv_0\}$

$\{r_2.av_0\} \overset{r_2|_{mv_0}}{\rightsquigarrow} \{r_2.cv_0\}$

$ubVarCard(r_2.ova_{ca}, ae_c, r_2) \overset{\text{Tab. 4.2.17, } (ix)}{=} ova_{ca}|_{cardv} = 1$

$dependsOn(r_5.av_1, r_5) = \{r_5.csubv_0, r_5.av_0\}$
$dependsOn(r_5.csubv_1, r_5) = \{r_5.csubv_0\}$

$\{r_5.csubv_0, r_5.av_0\} \overset{r_5|_{mv_0}}{\rightsquigarrow} \{r_5.csubv_0\}$

$ubVarCard(r_5.ova_{ca}, ae_c, r_5) \overset{\text{Tab. 4.2.17, } (ix)}{=} ova_{ca}|_{cardv} = 1$

Therefore it holds: $\sum_{ova, r_i : \ldots} ubVarCard(ova, ae_c, r_i) \leq 1$

With a similar argumentation the upper bounds conformance for the association between Attribute and Type, and for the key association between Class and Attribute can be shown.

Composition between ClassAssociation and ClassAssociationEnd The second association we analyze is the composition relation between ClassAssociation and ClassAssociationEnd. Each object of type ClassAssociation consists of at most two ClassAssociationEnds. Each object of type ClassAssociationEnd is contained in at most one ClassAssociation.

We begin with the simple case. The outgoing association from ClassAssociationEnd to ClassAssociation starts always at an object variable with the identifier $\diamondsuit$. So according to Theorem 4.2.2 the sum of all outgoing object variable associations of the regarding type has to be calculated. Within r there is always exactly one outgoing association with the cardinality of one. So it holds that $1 = ub \geq \sum ova'|_{cardv} = 1$ for all regarding outgoing associations within the rules r_4, r_7, r_8, r_9, r_{10}.

The case of the outgoing associations from ClassAssociation to ClassAssociationEnd is a little bit more complex. This case can be divided into two sub cases. First there is the situation where the object variable of type ClassAssociationEnd has an identifier $\diamondsuit$ assigned (rules: r_7, r_8, r_{10}). With the same argumentation then above it holds: $2 = ub \geq \sum ova'|_{cardv} = 2$ as there are always two ClassAssociationEnds connected to the regarding ClassAssociations.

Second in the rules r_4 and r_9 the object variables of the type ClassAssociation has an identifier with a value different to $\diamondsuit$. As there are no similar object variables with the same type and $r_4.cae v_1 \nsim r_9.cae v_1$ both sums according to Theorem 4.2.2 consist of exactly one summand. Let $r_4.ova_{aae}$ and $r_9.ova_{aae}$ be the according object variable associations, and let be $r_4.ae_{ae}$ and $r_9.ae_{ae}$ be the class association ends with the role names ae. So we have to calculate:

$ubVarCard(r_4.ova_{aae}, r_4.ae_{ae}, r_4)$
$ubVarCard(r_9.ova_{aae}, r_9.ae_{ae}, r_9)$

Both values may not succeed the upper bound 2.

It holds:

$dependsOn(r_4.cav_1, r_4) = \{r_4.cav_0\}$
$dependsOn(r_4.cae_1, r_4) = \diamondsuit$
$dependsOn(r_9.cav_1, r_9) = \{r_9.cav_0, r_9.bcv_0\}$
$dependsOn(r_9.cae_1, r_9) = \diamondsuit$

In both cases we have to calculate $maxmatch(r_i|_{mv_0}|_{mm}, r_i|_{mv_0}, M_0) \cdot ova|_{cardv}$. As the cardinality of the object associations is 1 and *maxmatch* equals to 2 when the class association of the left hand side (and the sub class) are considered as "fix" both *ubVarCard* values are equal to 2.

Upper bound conformance of r As for all other object variable associations on the right hand sides also Theorem 4.2.2 holds, which can be easily proven by similar deliberations the rule set r is upper bounds conform.

Metamodel conformance of r

As we have shown above the lower bounds conformance and the upper bounds conformance of r we can conclude with Theorem 4.2.6 that the rule set r is metamodel conform. That means for every input model with inheritance the rule set r transforms this model to a valid model according to the core metamodel mm_{core}.

5.3 BOTL Mapping to the UML

Though there is a UML-based notation for BOTL, which was presented in Section 3.3, so far the formalism allows us to transform only models that base on the notion of a model that comes with BOTL. In practice models are usually defined in terms of the UML metamodel. Thus, if the BOTL formalism should be applicable in practice then it must be possible to specify a precise mapping to the UML. In this section we introduce how such a mapping can be realized in terms

of BOTL rules. Therefore we determine how BOTL metamodels are mapped to UML class diagrams. Therefore we provide a set of BOTL rules that transform class diagrams in terms of the BOTL metamodel into those that conform to the UML metamodel.

5.3.1 Transformation Rules

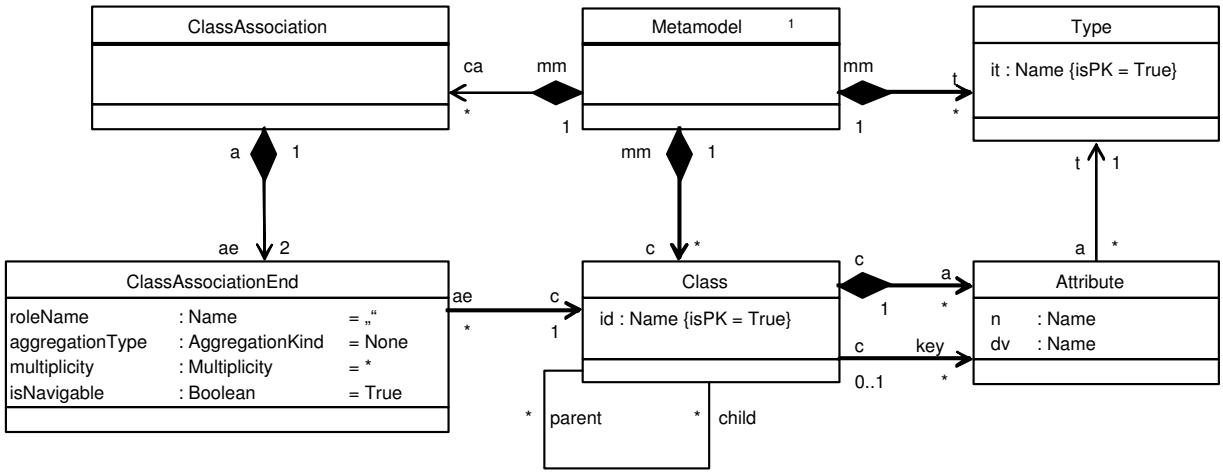

Figure 5.16: The BOTL metamodel in terms of a UML class diagram.

Figure 5.16 shows the already introduced BOTL metamodel (cf. Sec. 5.1). In Figure 5.16 pure UML instead of the UML profile defined in Section 3.3 is used. So instead of bold face, tagged values are used for marking attributes as primary keys. Also some default values for some attributes are given. Please note that we use an inheritance association with different semantics than in the latter section. This association indicates only a direct inheritance while the one presented in Section 5.2 builds the transitive hull. This is necessary to allow a BOTL-based transformation of inheritance structure (c.f. also rule r_5 in this section).

The metamodel for UML class diagrams is defined in the UML specification. We refer to the current version 1.4 of the standard [OMG02]. Since the entire metamodel of the UML is rather large and contains a lot of aspects that are not relevant for our transformation issues we present the part of the model that is relevant for our purposes. This subset of UML meta classes and associations was yielded by the following principles:

- Whenever information from a class symbol or a association between classes is mapped to meta classes of the UML meta model, these classes are regarded as “relevant”.
- All classes that associations with a lower bound different to 0 to the given classes are regarded as relevant, too.
- All classes that are (direct or indirect) parents of the given set of classes are also regarded as relevant.

Figure 5.17 depicts the (consistent) fragment of the UML metamodel that contains all classes and attributes that are relevant for the following transformation rules.

For the following proves we use the naming conventions for UML meta associations listed in Table 5.3.1.

With help of the following transformation rules BOTL metamodels can be mapped to the content that is generally represented by UML class diagrams.

Rule r_0 that is depicted in Figure 5.18 creates an instance of a UML DataType for every Type instance in the BOTL metamodel and vice versa. Thereby the names of the types are equal and determine their identity. Since the BOTL class Metamodel is a singleton the metamodel object is always the same. Therefore we do not have to consider this object in this and the subsequent rules. As one can see the UML attributes isRoot, isLeaf, isAbstract, and isActive are not mapped bijectively, because they are out of the scope of the BOTL transformation mechanism.

Analogously rule r_1 in Figure 5.19 maps BOTL instances of the type Class to UML Classes. Again irrelevant attributes are not part of the mapping and are denoted with a $\diamondsuit$.

Rule r_2 (cf. Fig. 5.20) maps every BOTL attribute of a class to an according UML class attribute. If the given class or data type does already exist the merge operator ensures that it is not created twice.

The primary key associations of BOTL metamodels are mapped to isPK tags of the UML representation with rule r_3 (cf. Fig. 5.21).

Rule r_4 (cf. Fig. 5.22) generates an UML association with one end for every pair of association/end in the BOTL notation. Thus for every association the rule matches two times - once for every end of the association.

Rule r_5 in Figure 5.23 translates inheritance relations among BOTL classes into those among UML classes. Please note that we use a different inheritance association here than in Section 5.2: The association with the role names parent and child indicates that the child class inherits *directly* from the parent class. The super-sub relations from Section 5.2 exists also among two classes where one might only inherit indirectly from the other. In fact it is necessary to calculate the latter one from the first one in a concrete BOTL metamodel if one wants to use the inheritance extension from Section 5.2. This calculation cannot be performed by BOTL rules since rule-based transformation languages (such as graph grammars, too) are not capable to calculate transitive hulls.

5.3.2 Applicability

Since we will need it for the proof of the applicability and metamodel conformance we first summarize the results of *dependsOn* for the given rule set in Table 5.4.

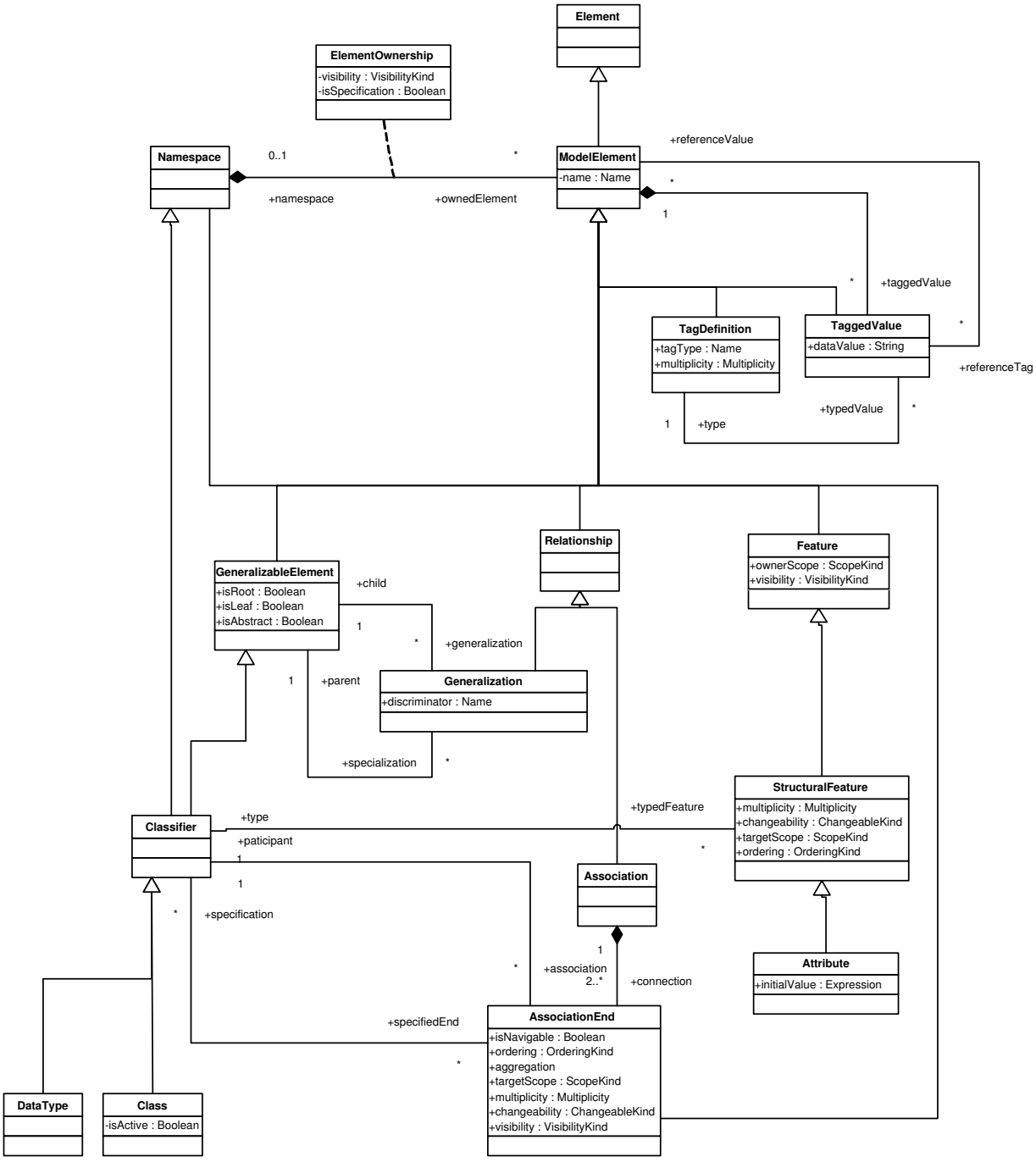

Figure 5.17: The fragment of the UML metamodel that is relevant for our transformations.

Name	Refers to
NOA_1	Class association between Namespace and ModelElement [a]
$NOAC_1$	Class association end at Namespace of NOA_1
$NOAA_1$	Class association end at ModelElement of NOA_1
TTA_1	Class association between StructuralFeature and Classifier [b]
$TTAA_1$	Class association end at typedFeature of TTA_1
$TTAT_1$	Class association end at type of TTA_1
AAA_1	Class association between Association and AssociationEnd
$AAAA_1$	Class association end at Association of AAA_1
$AAAE_1$	Class association end at AssociationEnd of AAA_1
PSA_1	Class association between GeneralizableElement and Generalization with the roles parent and specialization [c]
$PSAC_1$	Class association end at parent of PSA_1
$PSAC_1$	Class association end at specialization of PSA_1
GCA_1	Class association between GeneralizableElement and Generalization with the roles generalization and child [d]
$GCAC_1$	Class association end at child of GCA_1
$GCAG_1$	Class association end at generalizatin of GCA_1
NOA_1	Class association between Namespace and ModelElement [e]
$NOAC_1$	Class association end at Namespace of NOA_1
$NOAA_1$	Class association end at ModelElement of NOA_1
TA_1	Class association between ModelElement and TaggedValue [f]
TAA_1	Class association end at ModelElement of TA_1
TAT_1	Class association end at TaggedValue of TA_1
VDA_1	Class association between TagDefinition and TaggedValue
$VDAD_1$	Class association end at TagDefinition of VDA_1
$VDAV_1$	Class association end at TaggedValue of VDA_1
PAA_1	Class association between Classifier and AssociationEnd [g]
$PAAC_1$	Class association end at Classifier of PAA_1
$PAAA_1$	Class association end at AssociationEnd of PAA_1

Table 5.3: Naming conventions for the class associations of the UML metamodel

[a] At the instance level this association occurs between objects of the types Class and Attribute
[b] At the instance level this association occurs between objects of the types Attribute and DataType
[c] At the instance level this association occurs between objects of the types Class and Generalization
[d] At the instance level this association occurs between objects of the types Class and Generalization
[e] At the instance level this association occurs between objects of the types Class and Attribute
[f] At the instance level this association occurs between objects of the types Attribute and TaggedValue
[g] At the instance level this association occurs between objects of the types Class and AssociationEnd

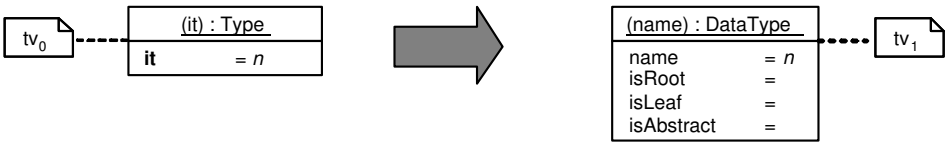

Figure 5.18: Rule r_0 generates a UML type for every BOTL type.

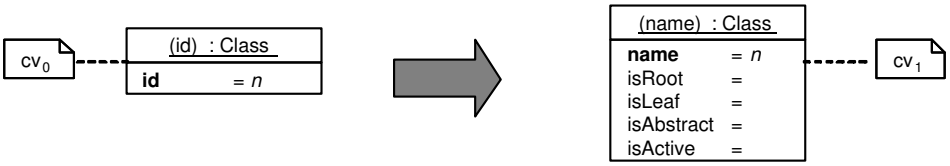

Figure 5.19: Rule r_1 generates a UML class for every BOTL class.

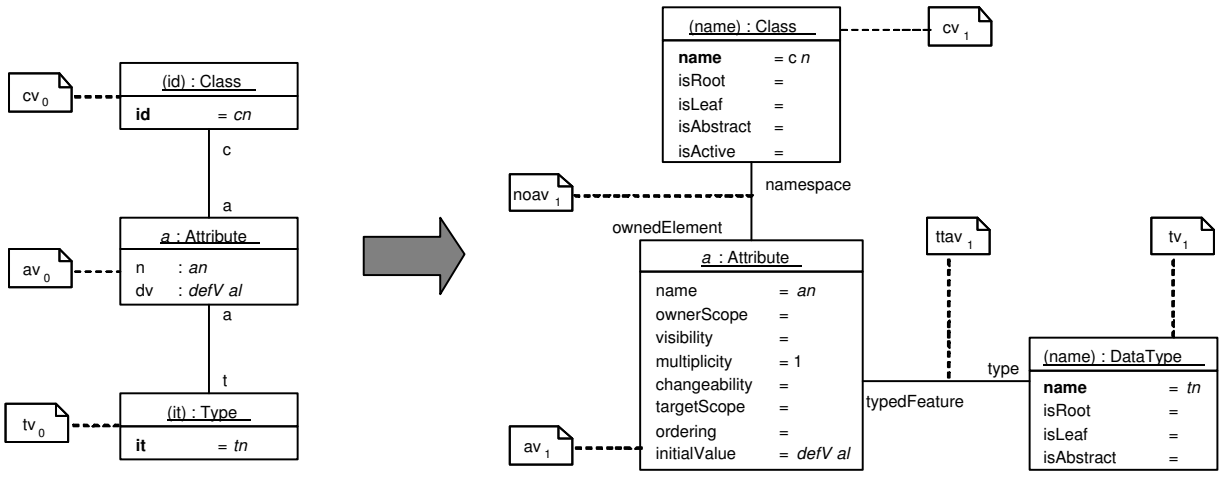

Figure 5.20: Rule r_2 generates a UML attribute for every BOTL attribute.

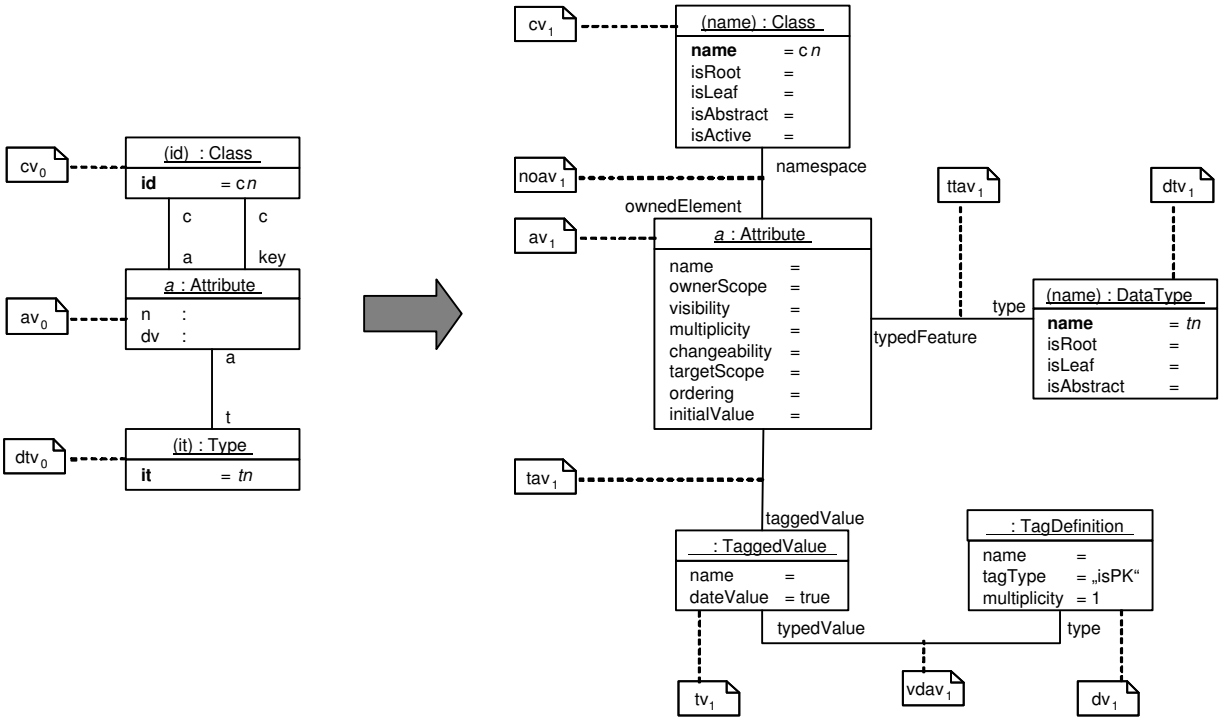

Figure 5.21: Rule r_3 relates the UML isPK tags with the according association of the UML metamodel.

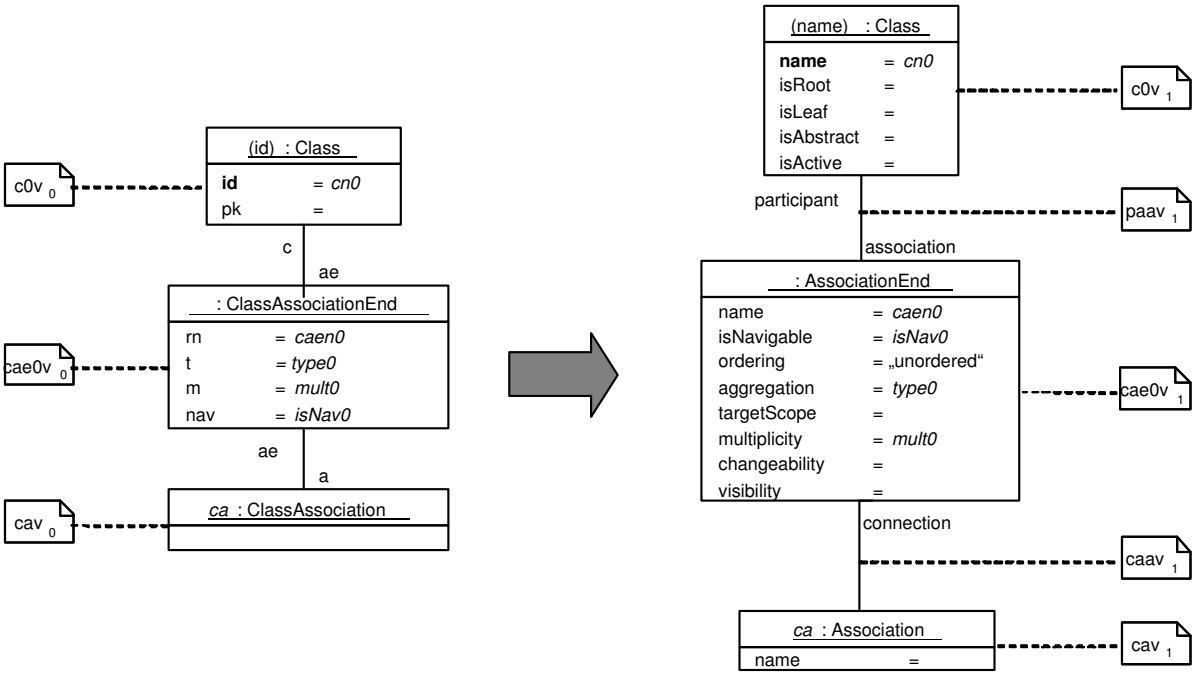

Figure 5.22: Rule r_4 generates associations between classes.

ov	r_i	$dependsOn(ov, r_i)$
$r_0.tv_1$	r_0	$\{r_0.tv_0\}$
$r_1.cv_1$	r_1	$\{r_1.cv_0\}$
$r_2.cv_1$	r_2	$\{r_2.cv_0\}$
$r_2.tv_1$	r_2	$\{r_2.tv_0\}$
$r_2.av_1$	r_2	$\{r_2.av_0\}$
$r_3.cv_1$	r_3	$\{r_3.cv_0\}$
$r_3.av_1$	r_3	$\{r_3.av_0\}$
$r_3.dtv_1$	r_3	$\{r_3.dtv_0\}$
$r_3.tv_1$	r_3	$\diamondsuit$
$r_3.dv_1$	r_3	$\diamondsuit$
$r_4.c0v_1$	r_4	$\{r_4.c0v_0\}$
$r_4.cae0v_1$	r_4	$\diamondsuit$
$r_4.cav_1$	r_4	$\{r_4.cav_0\}$
$r_5.c0v_1$	r_5	$\{r_5.c0v_0\}$
$r_5.gv_1$	r_5	$\diamondsuit$
$r_5.c1v_1$	r_5	$\{r_5.c1v_0\}$

Table 5.4: Overview over the results of $dependsOn(ov, r_i)$ for the given rule set.

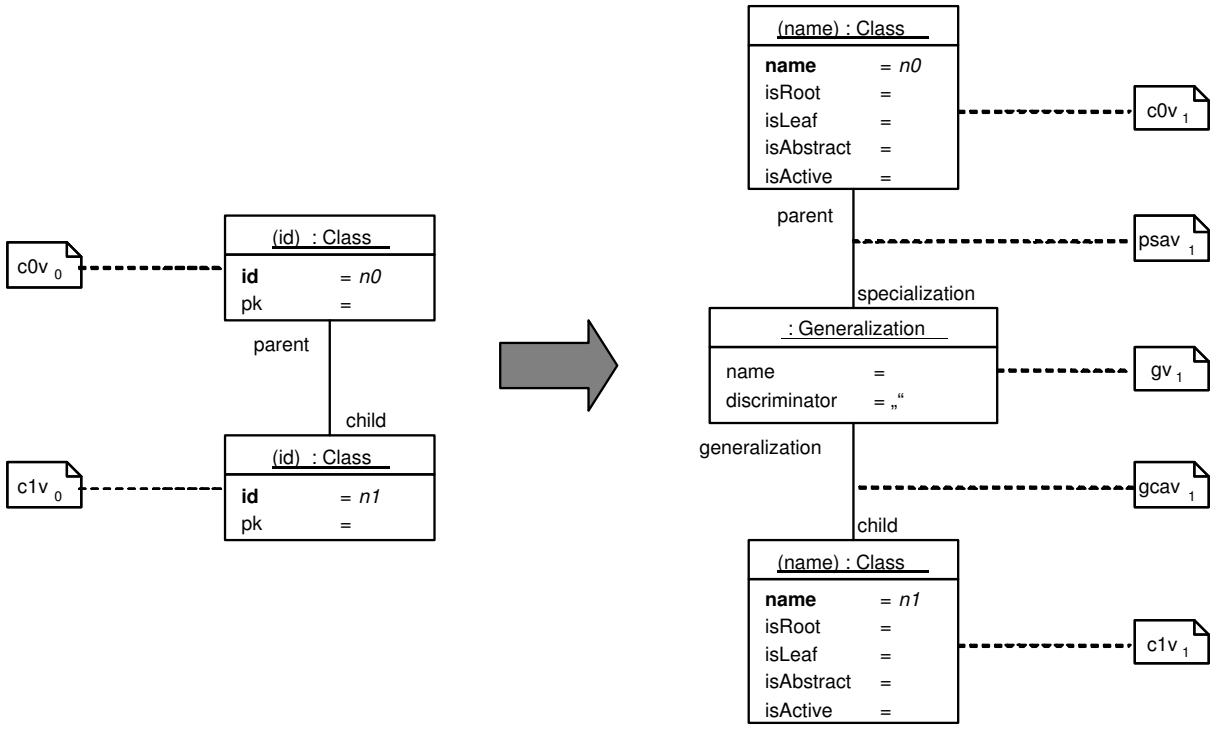

Figure 5.23: Rule r_5 generates generalization associations between classes.

According to Theorem 4.1.3 (i) it has to hold that every rule of the rule set is applicable. Thus we prove that every single rule is applicable.

Applicability of r_0 :

Let tv_0 be the object variable of the type Type at the rules left side. Further let tv_1 be the object variable at the rule's right side.

Verification of statement (i) in Theorem 4.1.2:

$$
\begin{aligned}
GL^0_{id}: \quad & (match^0_o(tv_0)|oi && = \varepsilon \quad \text{not considered according to Def. 3.4.5}) \\
GL^0_v: \quad & match^0_o(tv_0).it && = n \\
GL^1_{id}: \; & pK(match^1_o(tv_1).name) && = match^1_o(tv_1)|oi \\
GL^1_v: \quad & n && = match^1_o(tv_1).name \\
& \diamondsuit && = match^1_o(tv_1).isRoot \\
& \diamondsuit && = match^1_o(tv_1).isLeaf \\
& \diamondsuit && = match^1_o(tv_1).isAbstract
\end{aligned}
$$

Thus we can resolve the equational system and get an unambiguous solution for the right object variable's attributes its identifier:

$$\begin{aligned} match^1_o(tv_1)|oi &= pK(match^0_o(tv_0).n) \\ match^1_o(tv_1).name &= match^0_o(tv_0).n \\ match^1_o(tv_1).isRoot &= \diamondsuit \\ match^1_o(tv_1).isLeaf &= \diamondsuit \\ match^1_o(tv_1).isAbstract &= \diamondsuit \end{aligned}$$

Verification of statement (ii) in Theorem 4.1.2: Obviously it does hold:

$dependsOn(tv_1, r_0) = \{tv_0\}$

Further we can easily verify that $match^1_o(tv_1)$ always gets assigned the same attribute values when $match^0_o(tv_0)$ is constant. Thus the first part of 4.1.5 (ii) is satisfied.

Verification of statement (iii) in Theorem 4.1.2: The part (iii) of Theorem 4.1.2 is only relevant to right hand rule sides that contain at least two object variables of the same time and therefore isn't relevant for r_0.

Concluding we can state that rule r_0 is applicable.

Applicability of r_1 :

Let cv_0 be the object variable of the type Class at the rules left side. Further let cv_1 be the object variable at the rule's right side.

Verification of statement (i) in Theorem 4.1.2:

$$\begin{aligned} GL^0_{id}: \quad & (match^0_o(cv_0)|oi && = \varepsilon \quad \text{not considered according to Def. 3.4.5)} \\ GL^0_v: \quad & match^0_o(cv_0).id && = n \\ GL^1_{id}: \; & pK(match^1_o(cv_1).name) && = match^1_o(cv_1)|oi \\ GL^1_v: \quad & n && = match^1_o(cv_1).name \\ & \diamondsuit && = match^1_o(cv_1).isRoot \end{aligned}$$

$$\begin{aligned}\diamondsuit &= match_o^1(cv_1).isLeaf\\ \diamondsuit &= match_o^1(cv_1).isAbstract\\ \diamondsuit &= match_o^1(cv_1).isActive\end{aligned}$$

Thus we can resolve the equational system and get an unambiguous solution for the right object variable's attributes its identifier:

$$\begin{aligned}match_o^1(dtv)|oi &= pK(match_o^0(tv).n)\\ match_o^1(dtv).name &= match_o^0(tv).n\\ match_o^1(dtv).isRoot &= \diamondsuit\\ match_o^1(dtv).isLeaf &= \diamondsuit\\ match_o^1(dtv).isAbstract &= \diamondsuit\\ match_o^1(dtv).isActive &= \diamondsuit\end{aligned}$$

Verification of statement (ii) in Theorem 4.1.2: Obviously it does hold:

$dependsOn(cv_1, r_1) = \{cv_0\}$

Further we can easily verify that $match_o^1(cv_1)$ always gets assigned the same attribute values when $match_o^0(cv_0)$ is constant. Thus 4.1.2 (ii) is satisfied.

Verification of statement (iii) in Theorem 4.1.2: 4.1.2 (iii) is only relevant to right hand rule sides that contain at least two object variables of the same time and therefore isn't relevant for r_1.

Concluding we can state that rule r_1 is applicable.

Applicability of r_2 :

Again we we refer to object variables according to the identifiers in the UML comments.

Verification of statement (i) in Theorem 4.1.2:

$$
\begin{array}{rrll}
GL^0_{id}: (& match^0_o(cv_0)|oi & = \varepsilon & \text{not considered according to Def. 3.4.5)} \\
 & match^0_o(av_0)|oi & = a & \\
(& match^0_o(tv_0)|oi & = \varepsilon & \text{not considered according to Def. 3.4.5)} \\
GL^0_v: & match^0_o(cv_0).id & = cn & \\
 & match^0_o(av_0).n & = an & \\
 & match^0_o(av_0).dv & = defVal & \\
 & match^0_o(tv_0).it & = tn & \\
GL^1_{id}: & pK(\{(name, match^1_o(cv_1).name)\}) & = match^1_o(cv_1)|oi & \\
 & a & = match^1_o(av_1)|oi & \\
 & pK(\{(name, match^1_o(tv_1).name)\}) & = match^1_o(tv_1)|oi & \\
GL^1_v: & cn & = match^1_o(cv_1).name & \\
 & \diamondsuit & = match^1_o(cv_1).isRoot & \\
 & \diamondsuit & = match^1_o(cv_1).isLeaf & \\
 & \diamondsuit & = match^1_o(cv_1).isAbstract & \\
 & \diamondsuit & = match^1_o(cv_1).isActive & \\
 & an & = match^1_o(av_1).name & \\
 & \diamondsuit & = match^1_o(av_1).ownerScope & \\
 & \diamondsuit & = match^1_o(av_1).visibility & \\
 & 1 & = match^1_o(av_1).multiplicity & \\
 & \diamondsuit & = match^1_o(av_1).changeability & \\
 & \diamondsuit & = match^1_o(av_1).targetScope & \\
 & \diamondsuit & = match^1_o(av_1).ordering & \\
 & defVal & = match^1_o(av_1).initialValue & \\
 & tn & = match^1_o(tv_1).name & \\
 & \diamondsuit & = match^1_o(tv_1).isRoot & \\
 & \diamondsuit & = match^1_o(tv_1).isLeaf & \\
 & \diamondsuit & = match^1_o(tv_1).isAbstract &
\end{array}
$$

Thus we can resolve the equational system and get an unambiguous solution for the right object variables' attributes their identifiers:

$$
\begin{aligned}
match^1_o(cv_1)|oi &= pK(\{(name, match^0_o(cv_0).id)\}) \\
match^1_o(av_1)|oi &= match^0_o(av_0)|oi
\end{aligned}
$$

$$
\begin{aligned}
match^1_o(tv_1)|oi &= pK(\{(name, match^0_o(tv_0).id)\}) \\
match^1_o(cv_1).name &= match^0_o(cv_0).id \\
match^1_o(cv_1).isRoot &= \diamondsuit \\
match^1_o(cv_1).isLeaf &= \diamondsuit \\
match^1_o(cv_1).isAbstract &= \diamondsuit \\
match^1_o(cv_1).isActive &= \diamondsuit \\
match^1_o(av_1).name &= match^0_o(av_0).n \\
match^1_o(av_1).ownerScope &= \diamondsuit \\
match^1_o(av_1).visibility &= \diamondsuit \\
match^1_o(av_1).multiplicity &= 1 \\
match^1_o(av_1).changeability &= \diamondsuit \\
match^1_o(av_1).targetScope &= \diamondsuit \\
match^1_o(av_1).ordering &= \diamondsuit \\
match^1_o(av_1).initialValue &= match^0_o(av_0).dv \\
match^1_o(tv_1).name &= match^0_o(tv_0).it \\
match^1_o(tv_1).isRoot &= \diamondsuit \\
match^1_o(tv_1).isLeaf &= \diamondsuit \\
match^1_o(tv_1).isAbstract &= \diamondsuit
\end{aligned}
$$

Verification of statement (ii) in Theorem 4.1.2: It holds for cv_1:

$$
\begin{aligned}
dependsOn(cv_1, r_2) &= \{cv_0\} \\
attDependsOn(cv_1, (x, cv_1.x), r_2) &= \begin{cases} \{cv_0\} & \text{for } x = name \\ \emptyset & \text{for } cv_1|_{VV|a} \ni x \neq name \end{cases}
\end{aligned}
$$

Thus it does hold:

$$\forall a \in cv_1|VV|a : dependsOn(cv_1, r_2) \overset{r_2|_{mv_0}}{\rightsquigarrow} attDependsOn(cv_1, a, r_2)$$

It holds for av_1:

$$
\begin{aligned}
dependsOn(av_1, r_2) &= \{av_0\} \\
attDependsOn(av_1, (x, av_1.x), r_2) &= \begin{cases} \{av_0\} & \text{for } x \in \{name, initialValue\} \\ \emptyset & \text{for } av_1|_{VV|a} \ni x \notin \{name, initialValue\} \end{cases}
\end{aligned}
$$

Thus it does hold:

$$\forall a \in av_1|VV|a : dependsOn(av_1, r_2) \overset{r_2|_{mv_0}}{\rightsquigarrow} attDependsOn(av_1, a, r_2)$$

It holds for tv_1:

$$dependsOn(tv_1, r_2) = \{tv_0\}$$

$$attDependsOn(tv_1, (x, tv_1.x), r_2) = \begin{cases} \{tv_0\} & \text{for } x = name \\ \emptyset & \text{for } tv_1|_{VV|a} \ni x \neq name \end{cases}$$

Thus it does hold:

$$\forall a \in tv_1|VV|a : dependsOn(tv_1, r_2) \overset{r_2|_{mv_0}}{\rightsquigarrow} attDependsOn(tv_1, a, r_2)$$

Verification of statement (iii) in Theorem 4.1.2: Since there are no two object variables of the same type in $r_2|mv_1$ Theorem 4.1.2 (iii) does not apply and therefore r_2 is applicable.

Applicability of r_3 :

Since rule r_3 contains already eight different object variables we refer to them according to the identifiers in the UML comments.

Verification of statement (i) in Theorem 4.1.2:

$$\begin{array}{rrll}
GL^0_{id}: (& match^0_o(cv_0)|oi & = \varepsilon & \text{not considered according to Def. 3.4.5)} \\
 & match^0_o(av_0)|oi & = a & \\
(& match^0_o(dtv_0)|oi & = \varepsilon & \text{not considered according to Def. 3.4.5)} \\
GL^0_v: & match^0_o(cv_0).id & = cn & \\
 & match^0_o(av_0).n & = \diamondsuit & \\
(& match^0_o(av_0).dv & = \diamondsuit & \text{not considered according to Def. 3.4.5)} \\
 & match^0_o(dtv_0).it & = tn & \\
GL^1_{id}: & pK(\{(name, match^1_o(cv_1).name)\}) & = match^1_o(cv_1)|oi & \\
 & a & = match^1_o(av_1)|oi & \\
 & genID(2, v_3, genNum(tv_1, r_3|_{mv_1|OVB})) & = match^1_o(tv_1)|oi & \\
 & genID(2, v_3, genNum(dv_1, r_3|_{mv_1|OVB})) & = match^1_o(dv_1)|oi & \\
 & pK(\{(name, match^1_o(tv_1).name)\}) & = match^1_o(dtv_1)|oi &
\end{array}$$

$$
\begin{array}{lcl}
GL_v^1: & cn & = match_o^1(cv_1).name \\
& \diamondsuit & = match_o^1(cv_1).isRoot \\
& \diamondsuit & = match_o^1(cv_1).isLeaf \\
& \diamondsuit & = match_o^1(cv_1).isAbstract \\
& \diamondsuit & = match_o^1(cv_1).isActive \\
& \diamondsuit & = match_o^1(av_1).name \\
& \diamondsuit & = match_o^1(av_1).ownerScope \\
& \diamondsuit & = match_o^1(av_1).visibility \\
& \diamondsuit & = match_o^1(av_1).multiplicity \\
& \diamondsuit & = match_o^1(av_1).changeability \\
& \diamondsuit & = match_o^1(av_1).targetScope \\
& \diamondsuit & = match_o^1(av_1).ordering \\
& \diamondsuit & = match_o^1(av_1).initialValue \\
& \diamondsuit & = match_o^1(tv_1).name \\
& \text{true} & = match_o^1(tv_1).dateValue \\
& \diamondsuit & = match_o^1(dv_1).name \\
& \text{"isPK"} & = match_o^1(dv_1).tagType \\
& 1 & = match_o^1(dv_1).multiplicity \\
& tn & = match_o^1(dtv_1).name \\
& \diamondsuit & = match_o^1(dtv_1).isRoot \\
& \diamondsuit & = match_o^1(dtv_1).isLeaf \\
& \diamondsuit & = match_o^1(dtv_1).isAbstract
\end{array}
$$

Thus we can resolve the equational system and get an unambiguous solution for the right object variables' attributes their identifiers:

$$
\begin{aligned}
match_o^1(cv_1)|oi &= pK(\{(name, match_o^0(cv_0).id)\}) \\
match_o^1(av_1)|oi &= match_o^0(av_0)|oi \\
match_o^1(tv_1)|oi &= genID(2, v_3, genNum(tv_1, r_3|_{mv_1|OVB})) \\
match_o^1(dv_1)|oi &= genID(2, v_3, genNum(dv_1, r_3|_{mv_1|OVB})) \\
match_o^1(dtv_1)|oi &= pK(\{(name, match_o^0(dtv_0).id)\}) \\
match_o^1(cv_1).name &= match_o^0(cv_0).id \\
match_o^1(cv_1).isRoot &= \diamondsuit \\
match_o^1(cv_1).isLeaf &= \diamondsuit
\end{aligned}
$$

$$
\begin{aligned}
match^1_o(cv_1).isAbstract &= \diamondsuit \\
match^1_o(cv_1).isActive &= \diamondsuit \\
match^1_o(av_1).name &= \diamondsuit \\
match^1_o(av_1).ownerScope &= \diamondsuit \\
match^1_o(av_1).visibility &= \diamondsuit \\
match^1_o(av_1).multiplicity &= \diamondsuit \\
match^1_o(av_1).changeability &= \diamondsuit \\
match^1_o(av_1).targetScope &= \diamondsuit \\
match^1_o(av_1).ordering &= \diamondsuit \\
match^1_o(av_1).initialValue &= \diamondsuit \\
match^1_o(tv_1).name &= \diamondsuit \\
match^1_o(tv_1).dateValue &= \text{true} \\
match^1_o(dv_1).name &= \diamondsuit \\
match^1_o(dv_1).tagType &= \text{"isPK"} \\
match^1_o(dv_1).multiplicity &= 1 \\
match^1_o(dtv_1).name &= match^0_o(dtv_0).it \\
match^1_o(dtv_1).isRoot &= \diamondsuit \\
match^1_o(dtv_1).isLeaf &= \diamondsuit \\
match^1_o(dtv_1).isAbstract &= \diamondsuit
\end{aligned}
$$

Thus Theorem 4.1.2 (i) is satisfied.

Verification of statement (ii) in Theorem 4.1.2: Obviously it does hold for cv_1:

$$
\begin{aligned}
dependsOn(cv_1, r_3) &= \{cv_0\} \\
attDependsOn(cv_1, (\text{"}name\text{"}, \text{"}cn\text{"}), r_3) &= \{cv_0\} \\
\forall x \in cv_1|_{VV} \setminus \{(\text{"}name\text{"}, \text{"}cn\text{"})\} : attDependsOn(cv_1, (x, cv_1.x), r_3) &= \emptyset
\end{aligned}
$$

Thus it does hold:

$$\forall a \in cv_1|VV|a : dependsOn(cv_1, r_3) \overset{r_3|_{mv_0}}{\rightsquigarrow} attDependsOn(cv_1, a, r_3)$$

So Theorem 4.1.2 (ii) holds for cv_1.

It holds for av_1:

$$\forall (a,t) \in av_1|VV : t = \diamondsuit$$

So Theorem 4.1.2 (ii) holds for av_1.

It holds for dtv_1:

$$\begin{aligned} dependsOn(dtv_1, r_3) &= \{av_0\} \\ attDependsOn(av_1, (\text{"}name\text{"}, \text{"}tn\text{"}), r_3) &= \{av_0\} \\ \forall x \in dtv_1|_{VV} \setminus \{(\text{"}name\text{"}, \text{"}tn\text{"})\} : attDependsOn(cv_1, (x, cv_1.x), r_3) &= \emptyset \end{aligned}$$

Thus it does hold:

$$\forall a \in dtv_1|VV|a : dependsOn(dtv_1, r_3) \overset{r_3|_{mv_0}}{\rightsquigarrow} attDependsOn(dtv_1, a, r_3)$$

So Theorem 4.1.2 (ii) holds for dtv_1.

It holds for tv_1:

$$dependsOn(tv_1, r_3) = \diamondsuit$$

So Theorem 4.1.2 (ii) holds for tv_1.

It holds for tv_1:

$$dependsOn(dv_1, r_3) = \diamondsuit$$

So Theorem 4.1.2 (ii) holds for dv_1.

Verification of statement (iii) in Theorem 4.1.2: Since there are no two object variables of the same type in $r_3|mv_1$ Theorem 4.1.2 (iii) does not apply.

Thus r_3 is applicable.

Applicability of r_4 :

Verification of statement (i) in Theorem 4.1.2: To safe space we omit the equational system here and directly present its solution.

$$\begin{aligned} match^1_o(c0v_1)|oi &= pK(\{(name, match^0_o(c0v_0).id)\}) \\ match^1_o(cae0v_1)|oi &= genID(4, v_4, genNum(cae0v_1, r_4|_{mv_1|_{OVB}}) \\ match^1_o(cav_1)|oi &= match^0_o(cav_0)|oi \\ match^1_o(c0v_1).name &= match^0_o(c0v_0).id \end{aligned}$$

$$
\begin{aligned}
match^1_o(c0v_1).isRoot &= \diamondsuit \\
match^1_o(c0v_1).isLeaf &= \diamondsuit \\
match^1_o(c0v_1).isAbstract &= \diamondsuit \\
match^1_o(c0v_1).isActive &= \diamondsuit \\
match^1_o(cae0v_1).name &= match^0_o(cae0v_0).rn \\
match^1_o(cae0v_1).isNavigable &= match^0_o(cae0v_0).nav \\
match^1_o(cae0v_1).ordering &= \text{"unordered"} \\
match^1_o(cae0v_1).aggregation &= match^0_o(cae0v_0).t \\
match^1_o(cae0v_1).targetScope &= \diamondsuit \\
match^1_o(cae0v_1).multiplicity &= match^0_o(cae0v_0).m \\
match^1_o(cae0v_1).changeability &= \diamondsuit \\
match^1_o(cae0v_1).visibility &= \diamondsuit \\
match^1_o(cav_1).name &= \diamondsuit
\end{aligned}
$$

Please note that v_4 is a variable that is increased whenever r_4 is applied.

Because this solution is the unique solution of the rule's equational system 4.1.5 (i) is satisfied.

Verification of statement (ii) in Theorem 4.1.2: It holds for $c0v_1$:

$$
\begin{aligned}
dependsOn(c0v_1, r_4) &= \{c0v_0\} \\
attDependsOn(c0v_1, (x, c0v_1.x), r_4) &= \begin{cases} \{c0v_0\} & \text{for } x = name \\ \emptyset & \text{for } c0v_1|_{VV|a} \ni x \neq name \end{cases}
\end{aligned}
$$

Thus it does hold:

$$\forall a \in c0v_1|VV|a : dependsOn(c0v_1, r_4) \overset{r_4|_{mv_0}}{\rightsquigarrow} attDependsOn(c0v_1, a, r_4)$$

So Theorem 4.1.2 (ii) holds for $c0v_1$.

It holds for $cae0v_1$:

$$dependsOn(cae0v_1, r_4) = \diamondsuit$$

So Theorem 4.1.2 (ii) holds for $cae0v_1$.

It holds for cav_1: For cav_1 it holds:

$$
\begin{aligned}
dependsOn(cav_1, r_4) &= \{cav_0\} \\
attDependsOn(cav_1, (name, \diamondsuit), r_4) &= \emptyset
\end{aligned}
$$

Thus it does hold:

$$\forall a \in cav_1|VV|a : dependsOn(cav_1, r_4) \overset{r_4|_{mv_0}}{\rightsquigarrow} attDependsOn(cav_1, a, r_4)$$

So Theorem 4.1.2 (ii) holds for cav_1.

Thus (ii) does hold for the rule r_4.

Verification of statement (iii) in Theorem 4.1.2: Since there are no two object variables of the same type in $r_4|mv_1$ Theorem 4.1.2 (iii) does not apply.

Thus r_4 is applicable.

Applicability of r_5 :

Verification of statement (i) in Theorem 4.1.2: To safe space we omit the equational system here and directly present its solution.

$$\begin{aligned}
match^1_o(c0v_1)|oi &= pK(\{(name, match^0_o(c0v_0).id)\}) \\
match^1_o(c1v_1)|oi &= pK(\{(name, match^0_o(c1v_0).id)\}) \\
match^1_o(gv_1)|oi &= genID(5, v_5, genNum(gv_1, r_5|_{mv_1|_{OVB}}) \\
match^1_o(c0v_1).name &= match^0_o(c0v_0).id \\
match^1_o(c0v_1).isRoot &= \diamondsuit \\
match^1_o(c0v_1).isLeaf &= \diamondsuit \\
match^1_o(c0v_1).isAbstract &= \diamondsuit \\
match^1_o(c0v_1).isActive &= \diamondsuit \\
match^1_o(c1v_1).name &= match^0_o(c1v_0).id \\
match^1_o(c1v_1).isRoot &= \diamondsuit \\
match^1_o(c1v_1).isLeaf &= \diamondsuit \\
match^1_o(c1v_1).isAbstract &= \diamondsuit \\
match^1_o(c1v_1).isActive &= \diamondsuit \\
match^1_o(gv_1).name &= \diamondsuit \\
match^1_o(c1v_1).discriminator &= \text{""}
\end{aligned}$$

Please note that v_5 is a variable that is increased whenever r_5 is applied.

Because this solution is the unique solution of the rule's equational system 4.1.5 (i) is satisfied.

Verification of statement (ii) in Theorem 4.1.2: It holds for $c0v_1$:

$$dependsOn(c0v_1, r_5) = \{c0v_0\}$$
$$attDependsOn(c0v_1, (x, c0v_1.x), r_5) = \begin{cases} \{c0v_0\} & \text{for } x = name \\ \emptyset & \text{for } c0v_1|_{VV|a} \ni x \neq name \end{cases}$$

Thus it does hold:

$$\forall a \in c0v_1|VV|a : dependsOn(c0v_1, r_5) \overset{r_5|_{mv_0}}{\rightsquigarrow} attDependsOn(c0v_1, a, r_5)$$

So Theorem 4.1.2 (ii) holds for $c0v_1$.

It holds for $c1v_1$:

$$dependsOn(c1v_1, r_5) = \{c1v_0\}$$
$$attDependsOn(c1v_1, (x, c1v_1.x), r_5) = \begin{cases} \{c1v_0\} & \text{for } x = name \\ \emptyset & \text{for } c1v_1|_{VV|a} \ni x \neq name \end{cases}$$

Thus it does hold:

$$\forall a \in c1v_1|VV|a : dependsOn(c1v_1, r_5) \overset{r_5|_{mv_0}}{\rightsquigarrow} attDependsOn(c1v_1, a, r_5)$$

So Theorem 4.1.2 (ii) holds for $c1v_1$.

It holds for gv_1:

$$dependsOn(cae0v_1, r_4) = \diamondsuit$$

So Theorem 4.1.2 (ii) holds for $cae0v_1$.

Thus (ii) does hold for the rule r_5.

Verification of statement (iii) in Theorem 4.1.2: There are two object variables $c0v_1$ and $c1v_1$ in $r_5|_{mv_1}|_{OVB}$ for which it holds that $c0v_1|_{otv} = c1v_1|_{otv}$. Thus we regard the equational System CES:

$$CES:$$
$$mfm_k^1|_{match_o}(c0v_1)|_{oi} = mfm_l^1|_{match_o}(c1v_1)|_{oi}$$
$$GL'(c0v_1, c1v_1, k): \quad mfm_k^1|_{match_o}(c0v_1)|oi = pK(\{(name, mfm_k^0|_{match_o}(c0v_0).id)\})$$

$$
\begin{array}{lrl}
 & mfm_k^1|_{match_o}(c0v_1).name & = mfm_k^0|_{match_o}(c0v_0).id \\
 & mfm_k^1|_{match_o}(c0v_1).isRoot & = \diamondsuit \\
 & mfm_k^1|_{match_o}(c0v_1).isLeaf & = \diamondsuit \\
 & mfm_k^1|_{match_o}(c0v_1).isAbstract & = \diamondsuit \\
 & mfm_k^1|_{match_o}(c0v_1).isActive & = \diamondsuit \\
GL'(c0v_1, c1v_1, k): & mfm_l^1|_{match_o}(c0v_1)|oi & = pK(\{(name, mfm_l^0|_{match_o}(c0v_0).id)\}) \\
 & mfm_l^1|_{match_o}(c0v_1).name & = mfm_l^0|_{match_o}(c0v_0).id \\
 & mfm_l^1|_{match_o}(c0v_1).isRoot & = \diamondsuit \\
 & mfm_l^1|_{match_o}(c0v_1).isLeaf & = \diamondsuit \\
 & mfm_l^1|_{match_o}(c0v_1).isAbstract & = \diamondsuit \\
 & mfm_l^1|_{match_o}(c0v_1).isActive & = \diamondsuit
\end{array}
$$

Obviously *CES* has a solution for the case when

$mfm_k^0|_{match_o}(c0v_0).id = mfm_l^0|_{match_o}(c0v_0).id$

Since all attribute values of the two object variables $c0v_1$ and $c1v_1$ that are not key attributes have the value $\diamondsuit$ we can apply Lemma 4.1.3 to prove that statement (iii) does hold.

Thus r_5 is applicable.

Applicability of *r*

To prove the applicability of a rule set we use Theorem 4.1.1. Part (i) states that every single rule of *r* has to be applicable, which was already shown in the previous sections. Part (ii) states that for any two rules containing two object variables of the same type that have attribute values which are not primary keys at least one value must be $\diamondsuit$.

Table 5.3.2 subsumes the different attribute values that are generated by the rules. As one can easily verify 5.3.2 (ii) does hold here.

Concluding we can state that *r* is applicable.

5.3.3 Metamodel Conformance

We show that *r* is metamodel conform according to Theorem 4.2.6.

class attribute	r_0	r_1	r_2	r_3	r_4	r_5
DataType.name	(pK)	-	-	(pK)	-	-
DataType.isRoot	♢	-	♢	♢	-	-
DataType.isLeaf	♢	-	♢	♢	-	-
DataType.isAbstract	♢	-	♢	♢	-	-
Class.name	-	(pK)	(pK)	(pK)	(pK)	(pK)
Class.isRoot	-	♢	♢	♢	♢	♢
Class.isLeaf	-	♢	♢	♢	♢	♢
Class.isAbstract	-	♢	♢	♢	♢	♢
Class.isActive	-	♢	♢	♢	♢	♢
Attribute.name	-	-	*an*	♢	-	-
Attribute.ownerScope	-	-	♢	♢	-	-
Attribute.visibility	-	-	♢	♢	-	-
Attribute.multiplicity	-	-	"1"	♢	-	-
Attribute.changability	-	-	♢	♢	-	-
Attribute.targetScope	-	-	♢	♢	-	-
Attribute.ordering	-	-	♢	♢	-	-
Attribute.initialValue	-	-	$defVal$	♢	-	-
TaggedValue.name	-	-	-	♢	-	-
TaggedValue.dataValue	-	-	-	"true"	-	-
TagDefinition.name	-	-	-	♢	-	-
TagDefinition.tagType	-	-	-	"isPK"	-	-
TagDefinition.multiplicity	-	-	-	"1"	-	-
AssociationEnd.name	-	-	-	-	$caen0$	-
AssociationEnd.isNavigable	-	-	-	-	$isNav0$	-
AssociationEnd.ordering	-	-	-	-	"unordered"	-
AssociationEnd.aggregation	-	-	-	-	$type0$	-
AssociationEnd.targetScope	-	-	-	-	♢	-
AssociationEnd.multiplicity	-	-	-	-	$mult0$	-
AssociationEnd.changeability	-	-	-	-	♢	-
AssociationEnd.visibility	-	-	-	-	♢	-
Association.name	-	-	-	-	♢	-
Generalization.name	-	-	-	-	-	♢
Generalization.discriminator	-	-	-	-	-	" "

Table 5.5: The attribute values that are written by the different rules of r that are needed to prove applicability according to Theorem 4.1.1.

varLbConformance of the rules

According to Theorem 4.2.6 (i) we have to show therefore that r is *lbConform*.

r_0: Definition (4.2.22) holds obviously, because $r_0|_{mv_1}$ consists only of one object variable of the type DataType. In the UML metamodel the class DataType has no outgoing associations with lower bounds that are higher than 0. $r_0|mv_1$ doesn't contain any object variable associations.

Thus it holds $varLBConform(r_0|_{mv1}, r_0|_{mv1|_{mm}}) \overset{Lemma 4.2.8}{\Rightarrow} lbConform(r_0)$

r_1: Definition (4.2.22) holds obviously, because $r_1|_{mv_1}$ consists only of one object variable of the type Class. In the UML metamodel the class Class and its parent classes have no outgoing associations with lower bounds that are higher than 0.

Thus it holds $varLBConform(r_1|_{mv1}, r_1|_{mv1|_{mm}}) \overset{Lemma 4.2.8}{\Rightarrow} lbConform(r_1)$

r_2: In the UML metamodel the classes Class and DataType have are no outgoing associations with lower bounds that are higher than 0 (c.f. Figure 5.17). The class Attribute inherits one association form StructuralFeature that has an to-one association to Classifier. In $r_2|mv_1$ this lower bound (of 1) is satisfied, because there exists an association of the correct type between Attribute and DataType (which is a subclass of Classifier).

Thus it holds $varLBConform(r_2|_{mv1}, r_2|_{mv1|_{mm}}) \overset{Lemma 4.2.8}{\Rightarrow} lbConform(r_2)$

r_3: Again in the UML metamodel the class Class has no outgoing associations with lower bounds that are higher than 0.

There exists an outgoing association from Attribute to DataType (wich is inherited from StructuralFeature and Classifier) with a multiplicity of 1. Since this association is part of the model variable the Definition 4.2.22 is not violated.

There exists an outgoing association from TaggedValue to ModelElement with a with a multiplicity of 1. This association exists in the model variable between tv_1 and the TagDefinition dv_1.

There exists an outgoing association from TaggedValue to TagDefinition with a with a multiplicity of 1. This association exists in the model variable between tv_1 and the Attribute av_1, which inherits from ModelElement.

Thus it holds $varLBConform(r_3|_{mv1}, r_3|_{mv1|_{mm}}) \overset{Lemma 4.2.8}{\Rightarrow} lbConform(r_3)$

r_5: In the UML metamodel the class Class has no outgoing associations with lower bounds that are higher than 0.

A Generalization has an outgoing association to a child which is of the type GeneralizableElement with a multiplicity of 1. In the model variable there is an appropriate association to a Class (which inherits indirectly from GeneralizableElement) from gv_1 to $c1v_1$.

Further a Generalization has an outgoing association to a parent of the type GeneralizableElement with a multiplicity of 1. In the model variable there is an appropriate association to a Class (which inherits indirectly from the class GeneralizableElement) from gv_1 to $c0v_1$.

Thus r_5 is *varLbConform*. $\overset{Lemma 4.2.8}{\Rightarrow}$ $lbConform(r_5)$

According to Lemma 4.2.8 we know that the rules r_0, r_1, r_2, r_3, and r_5 are also *lbConform*.

Obviously rule r_4 isn't *varLbConform* because the UML metamodel contains an association from Association to Association with a lower bound of 2 that does not occur in r_4. Therefore we have to prove that r_4 is *lbConform* according to Theorem 4.2.4 (see page 105):

$$
\begin{aligned}
AAA_1 &:= \{(\text{"}connection\text{"}, AssociationEnd, (2,\infty), \text{"none"}, true),\\
&\quad (\text{""}, Association, (1,1), \text{"composite"}, true)\}\\
AAAA_1 &:= (\text{""}, Association, (1,1), \text{"composite"}, true)\}\\
AAAE_1 &:= (\text{"}connection\text{"}, AssociationEnd, (2,\infty), \text{"none"}, true)\\
caav_1 &:= (AAA_1, 1, \{(AAAE_1, cae0v_1), (AAAA_1, cav_1)\})\\
PAA_1 &:= \{(\text{"}association\text{"}, AssociationEnd, (0,\infty), \text{"none"}, true),\\
&\quad (\text{"}participant\text{"}, Classifier, (1,1), \text{"none"}, true)\}\\
PAAC_1 &:= (\text{"}participant\text{"}, Classifier, (1,1), \text{"none"}, true)\\
PAAA_1 &:= (\text{"}association\text{"}, AssociationEnd, (0,\infty), \text{"none"}, true)\\
paav_1 &:= (PAA_1, 1, \{(PAAC_1, c0v_1), (PAAA_1, cae0v_1)\})
\end{aligned}
$$

Obviously it holds:

$$
\begin{aligned}
dependsOn(cav_1, r_4) &= \{cav_0\}\\
dependsOn(cae0v_1, r_4) &= \diamondsuit\\
dependsOn(c0v_1, r_4) &= \{c0v_0\}
\end{aligned}
$$

Calculation of $lbCard(AAA_1, AAAA_1, r_4)$:

$$
\begin{aligned}
lbCard(AAA_1, AAAA_1, r_4) &\overset{Def.4.2.27}{=} \min_{ova:ova|_{OVAT}=AAA_1} lbVarCard(ova, AAAA_1, r_4) \\
&= lbVarCard(caav_1, AAAA_1, r_4) \\
&\overset{\text{Table 4.2.26, Case (xii)}}{=} caav_1|cardv \\
&= 1 \\
&\geq 1 \widehat{=} \text{ lower bound of } AAAA_1
\end{aligned}
$$

Calculation of $lbCard(AAA_1, AAAE_1, r_4)$:

$$
\begin{aligned}
lbCard(AAA_1, AAAE_1, r_4) &\overset{Def.4.2.27}{=} \min_{ova:ova|_{OVAT}=AAA_1} lbVarCard(ova, AAAE_1, r_4) \\
&= lbVarCard(caav_1, AAAE_1, r_4) \\
&\overset{\text{Table 4.2.26, Case (xvi)}}{=} minmatch(r_4|_{mv_0|_{mm}}, r_4|_{mv_0}, \{cav_0\}) \\
&= 2 \\
&\geq 2 \widehat{=} \text{ lower bound of } AAAE_1
\end{aligned}
$$

Calculation of $lbCard(PAA_1, PAAC_1, r_4)$:

$$
\begin{aligned}
lbCard(PAA_1, PAAC_1, r_4) &\overset{Def.4.2.27}{=} \min_{ova:ova|_{OVAT}=AAA_1} lbVarCard(ova, PAAC_1, r_4) \\
&= lbVarCard(paav_1, PAAC_1, r_4) \\
&\overset{\text{Table 4.2.26, Case (ix)}}{=} paav_1|cardv \\
&= 1 \\
&\geq 1 \widehat{=} \text{ lower bound of } PAAC_1
\end{aligned}
$$

Calculation of $lbCard(PAA_1, PAAA_1, r_4)$:

$$
\begin{aligned}
lbCard(PAA_1, PAAA_1, r_4) &\overset{Def.4.2.27}{=} \min_{ova:ova|_{OVAT}=AAA_1} lbVarCard(ova, PAAA_1, r_4) \\
&= lbVarCard(paav_1, PAAA_1, r_4)
\end{aligned}
$$

$$\begin{array}{lll} & \overset{\text{Table 4.2.26, Case (ix)}}{=} & paav_1|cardv \\ & = & 1 \\ & \geq & 0 \widehat{=} \text{ lower bound of } PAAA_1 \end{array}$$

Thus we know that $lbConform(r_4)$ holds according to Theorem 4.2.4.

Now we have proved that all rules of r are *lbConform*. Since we've already shown in Section 5.3.2 that r is applicable we can state according to Theorem 4.2.5 that r is *lbConform*.

ubConformance of the rules

Now we show that the postulation (i) of Theorem 4.2.6 does hold vor all association ends that occur in our rules.

Table 5.6 shows the results we obtain for $mv_\alpha \sqsubseteq mv_\beta$ when comparing our rules' left side model variables.

Table 5.7 shows the results we obtain for $OVP(r_\alpha, r_\beta)$ when comparing our rules for the BOTL-to-UML transformation. Please note that we qualify object variable names with their according rule name to be able to differ them. E.g. $r_i.ov$ denotes the object variable ov within the rule r_i.

Redundant associations in rule r_0: No associations

Redundant associations in rule r_1: No associations

Redundant associations in rule r_2: Calculation of $redundant(ova_i, r_2, r)$ according to Definition 4.2.19.

As we can see from Table 5.6 it holds:

$r_0|_{mv_0} \sqsubseteq r_2|_{mv_0} \quad \wedge$
$r_1|_{mv_0} \sqsubseteq r_2|_{mv_0}$

with (see table 5.7)

$OVP(r_2, r_0) = \{\{(r0.tv_1, r_2.tv_1)\}\}$
$OVP(r_2, r_1) = \{\{(r1.cv_1, r_2.cv_1)\}\}$

r_{sub}	r_{super}	$r_{sub}\|_{mv_0} \sqsubseteq r_{super}\|_{mv_0}$
$r_0\|_{mv_0}$	$r_1\|_{mv_0}$	false
$r_0\|_{mv_0}$	$r_2\|_{mv_0}$	true
$r_0\|_{mv_0}$	$r_3\|_{mv_0}$	true
$r_0\|_{mv_0}$	$r_4\|_{mv_0}$	false
$r_0\|_{mv_0}$	$r_5\|_{mv_0}$	false
$r_1\|_{mv_0}$	$r_0\|_{mv_0}$	false
$r_1\|_{mv_0}$	$r_2\|_{mv_0}$	true
$r_1\|_{mv_0}$	$r_3\|_{mv_0}$	true
$r_1\|_{mv_0}$	$r_4\|_{mv_0}$	true
$r_1\|_{mv_0}$	$r_5\|_{mv_0}$	true
$r_2\|_{mv_0}$	$r_0\|_{mv_0}$	false
$r_2\|_{mv_0}$	$r_1\|_{mv_0}$	false
$r_2\|_{mv_0}$	$r_3\|_{mv_0}$	true
$r_2\|_{mv_0}$	$r_4\|_{mv_0}$	false
$r_2\|_{mv_0}$	$r_5\|_{mv_0}$	false
$r_3\|_{mv_0}$	$r_0\|_{mv_0}$	false
$r_3\|_{mv_0}$	$r_1\|_{mv_0}$	false
$r_3\|_{mv_0}$	$r_2\|_{mv_0}$	false
$r_3\|_{mv_0}$	$r_4\|_{mv_0}$	false
$r_3\|_{mv_0}$	$r_5\|_{mv_0}$	false
$r_4\|_{mv_0}$	$r_0\|_{mv_0}$	false
$r_4\|_{mv_0}$	$r_1\|_{mv_0}$	false
$r_4\|_{mv_0}$	$r_2\|_{mv_0}$	false
$r_4\|_{mv_0}$	$r_3\|_{mv_0}$	false
$r_4\|_{mv_0}$	$r_5\|_{mv_0}$	false
$r_5\|_{mv_0}$	$r_0\|_{mv_0}$	false
$r_5\|_{mv_0}$	$r_1\|_{mv_0}$	false
$r_5\|_{mv_0}$	$r_2\|_{mv_0}$	false
$r_5\|_{mv_0}$	$r_3\|_{mv_0}$	false
$r_5\|_{mv_0}$	$r_4\|_{mv_0}$	false

Table 5.6: The results of $mv_{sub} \sqsubseteq mv_{super}$ for $mv_{sub}, mv_{super} \in r_i|mv_0$

r_{super}	r_{sub}	OVP($r_{super}r_{sub}$)
r_1	r_0	$\emptyset$
r_2	r_0	$\{\{(r0.tv_1, r_2.tv_1)\}\}$
r_3	r_0	$\{\{(r0.tv_1, r_3.dtv_1)\}\}$
r_4	r_0	$\emptyset$
r_5	r_0	$\emptyset$
r_0	r_1	$\emptyset$
r_2	r_1	$\{\{(r1.cv_1, r_2.cv_1)\}\}$
r_3	r_1	$\{\{(r1.cv_1, r_3.cv_1)\}\}$
r_4	r_1	$\{\{(r1.cv_1, r_4.c0v_1)\}$
r_5	r_1	$\{\{(r1.cv_1, r_5.c0v_1)\}, \{(r1.cv_1, r_5.c0v_1)\}\}$
r_0	r_2	$\emptyset$
r_1	r_2	$\emptyset$
r_3	r_2	$\{\{(r2.cv_1, r_3.cv_1), (r2.av_1, r_3.av_1), (r2.tv_1, r_3.dtv_1)\}\}$
r_4	r_2	$\emptyset$
r_5	r_2	$\emptyset$
r_0	r_3	$\emptyset$
r_1	r_3	$\emptyset$
r_2	r_3	$\emptyset$
r_4	r_3	$\emptyset$
r_5	r_3	$\emptyset$
r_0	r_4	$\emptyset$
r_1	r_4	$\emptyset$
r_2	r_4	$\emptyset$
r_3	r_4	$\emptyset$
r_5	r_4	$\emptyset$

Table 5.7: The results of $OVP(r_{super}, r_{sub})$.

$r_0|_{mv_1}$ and $r_1|_{mv_1}$ contain no association variables, thus there exists no ova_j as required by Definition 4.2.19 (ii). Therefor we can state:

$$redundant(r2.noav_1, r_2, r) = false$$
$$redundant(r2.ttav_1, r_2, r) = false$$

Redundant associations in rule r_3: Calculation of $redundant(ova_i, r_3, r)$ according to Definition 4.2.19.

As we can see from Table 5.6 it holds:

$$r_0|_{mv_0} \sqsubseteq r_3|_{mv_0} \quad \wedge$$
$$r_1|_{mv_0} \sqsubseteq r_3|_{mv_0} \quad \wedge$$
$$r_2|_{mv_0} \sqsubseteq r_3|_{mv_0}$$

with (see table 5.7)

$$OVP(r_3, r_0) = \{\{(r0.tv_1, r_3.dtv_1)\}\}$$
$$OVP(r_3, r_1) = \{\{(r1.cv_1, r_3.cv_1)\}\}$$
$$OVP(r_3, r_2) = \{\{(r2.cv_1, r_3.cv_1), (r2.av_1, r_3.av_1), (r2.tv_1, r_3.dtv_1)\}\}$$

Thus there exist the rules r_0, r_1, and r_2 that hold for Definition 4.2.19(i).

$r_0|_{mv_1}$ and $r_1|_{mv_1}$ contain no association variables, thus there exists no ova_j as required by Definition 4.2.19 (ii). $r_2|_{mv_1}$ contains two variable associations $r_2.noav_1$ and $r_2.ttav_1$.

$redundant(r_3.noav_1, r_3, r)$:
The first postulation in Definition 4.2.19 (ii) implies that an ova_j from $r_j|mv_1$ must be of the same type as ova_i. This holds only for $r_2.noav_1$, so let $r_2.noav_1$ be the association variable ova_j according to Definition 4.2.19 (ii). Obviously the statements of (ii) do hold, since the two object variables at the end of the association variables $r1.cv_1$ resp. $r_3.cv_1$ and $r2.av_1$ resp. $r_3.av_1$ are pairwise in $OVP(r_3, r_2)$. Further it holds that $1 = r_3.noav_1|_{cardv} \geq r_2.noav_1|_{cardv} = 1$. So it holds:

$$redundant(r_3.noav_1, r_3, r) = true$$

$redundant(r_3.ttav_1, r_3, r)$ $r_2|_{mv_1}$ contains only one variable association that is of the same type as $r_3.ttav_1$, which is $r_2.ttav_1$. So let $r_2.noav_1$ be the association variable ova_j according to Definition 4.2.19 (ii). Obviously the statements of (ii) do hold, since the two object variables at the end of the association variables $r1.tv_1$ resp. $r_3.dtv_1$ and $r2.av_1$ resp. $r_3.av_1$ are pairwise in $OVP(r_3, r_2)$. Further it holds that $1 = r_3.ttav_1|_{cardv} \geq r_2.ttav_1|_{cardv} = 1$. So it holds:

$$redundant(r_3.ttav_1, r_3, r) = true$$

As we've seen the other variables associations of $r_3|_{mv_1}$ must yield to fast. Concluding we obtain:

$$redundant(r_3.noav_1, r_3, r) = true$$
$$redundant(r_3.ttav_1, r_3, r) = true$$
$$redundant(r_3.tav_1, r_3, r) = false$$
$$redundant(r_3.vdav_1, r_3, r) = false$$

Redundant associations in rule r_4: Calculation of $redundant(ova_i, r_4, r)$ according to Definition 4.2.19.

As we can see from Table 5.6 it holds:

$$r_1|_{mv_0} \sqsubseteq r_4|_{mv_0}$$

with (see table 5.7)

$$OVP(r_4, r_1) = \{\{(r1.cv_1, r_4.c0v_1)\}\}$$

Thus there exists only the rule r_1 that holds for Definition 4.2.19(i).

$r_1|_{mv_1}$ contains no association variables, thus there exists no ova_j as required by Definition 4.2.19 (ii). Thus we can state

$$redundant(r_4.paav_1, r_4, r) = false$$
$$redundant(r_4.caav_1, r_4, r) = false$$

Redundant associations in rule r_5: Calculation of $redundant(ova_i, r_4, r)$ according to Definition 4.2.19.

As we can see from Table 5.6 it holds:

$$r_1|_{mv_0} \sqsubseteq r_5|_{mv_0}$$

with (see table 5.7)

$$OVP(r_5, r_1) = \{\{(r1.cv_1, r_5.c0v_1)\}, \{(r1.cv_1, r_5.c0v_1)\}\}$$

Thus there exists only the rule r_1 that holds for Definition 4.2.19(i).

$r_1|_{mv_1}$ contains no association variables, thus there exists no ova_j as required by Definition 4.2.19 (ii). Thus we can state

$$redundant(r_5.psav_1, r_5, r) = false$$
$$redundant(r_5.gcav_1, r_5, r) = false$$

The results of the evaluation of *redundant* are concluded in table 5.8.

ova	r_i	$redundant(ova, r_i, r)$
$r_2.noav_1$	r_2	false
$r_2.ttav_1$	r_2	false
$r_3.noav_1$	r_3	true
$r_3.ttav_1$	r_3	true
$r_3.tav_1$	r_3	false
$r_3.vdav_1$	r_3	false
$r_4.paav_1$	r_4	false
$r_4.caav_1$	r_4	false
$r_5.psav_1$	r_5	false
$r_5.gcav_1$	r_5	false

Table 5.8: Overview over the results of $redundant(ova, r_i, r)$ for all right hand association variables of the rule set.

We now can apply Theorem 4.2.2. The result of its application are listed in Table 5.9. As it turns out several values for *ubVarCard* have to be calculated. Together with the results of *dependsOn* (see Table 5.4) we can provide the following solutions that are necessary for the application of Theorem 4.2.2:

$$ubVarCard(r_2.noav_1, NOAA_1, r_2) \overset{\text{Tab. 4.2.17 (ix)}}{=} maxmatch(r_2|_{mv0}|_{mm}, r_2|_{mv_0}, cv_0) \cdot 1 = \infty$$
$$ubVarCard(r_2.noav_1, NOAC_1, r_2) \overset{\text{Tab. 4.2.17 (ix)}}{=} maxmatch(r_2|_{mv0}|_{mm}, r_2|_{mv_0}, av_0) \cdot 1 = 1$$
$$ubVarCard(r_2.ttav_1, TTAT_1, r_2) \overset{\text{Tab. 4.2.17 (ix)}}{=} maxmatch(r_2|_{mv0}|_{mm}, r_2|_{mv_0}, av_0) \cdot 1 = 1$$
$$ubVarCard(r_2.ttav_1, TTAA_1, r_2) \overset{\text{Tab. 4.2.17 (ix)}}{=} maxmatch(r_2|_{mv0}|_{mm}, r_2|_{mv_0}, tv_0) \cdot 1 = \infty$$
$$ubVarVard(r_3.tav_1, TAT_1, r_3) \overset{\text{Tab. 4.2.17 (xvi)}}{=} maxmatch(r_3|_{mv_0|_{mm}}, r_3|_{mv_0}, \{av_0\}) \cdot 1 = 1$$
$$ubVarCard(r_4.paav_1, PAAA_1, r_4) \overset{\text{Tab. 4.2.17 (xvi)}}{=} maxmatch(r_4|_{mv_0|_{mm}}, r_4|_{mv_0}, \{c0v_0\}) \cdot 1 = \infty$$
$$ubVarCard(r_4.caa0v_1, AAAE_1, r_4) \overset{\text{Tab. 4.2.17 (xvi)}}{=} maxmatch(r_4|_{mv_0|_{mm}}, r_4|_{mv_0}, \{cav_0\}) \cdot 1 = \infty$$
$$ubVarCard(r_5.psav_1, PSAG_1, r_5) \overset{\text{Tab. 4.2.17 (xvi)}}{=} maxmatch(r_5|_{mv_0|_{mm}}, r_5|_{mv_0}, \{c0v_0\}) \cdot 1 = \infty$$

$$ubVarCard(r_5.gcav_1, CGAG_1, r_5) \overset{\text{Tab. 4.2.17 (xvi)}}{=} maxmatch(r_5|_{mv_0|mm}, r_5|_{mv_0}, \{c1v_0\}) \cdot 1 = \infty$$

r_k	ov_0	ae_1	$ov_0\|_{oiv}$	$\sum$ of case 1	$\sum$ of case 2	ub
r_2	$r_2.cv_1$	$NOAA_1$	$\neq \Diamond$	-	$ubVarCard(r_2.noav_1, NOAA_1, r_2) = \infty$	∞
r_2	$r_2.av_1$	$NOAC_1$	$\neq \Diamond$	-	$ubVarCard(r_2.noav_1, NOAC_1, r_2) = 1$	1
r_2	$r_2.av_1$	$TTAT_1$	$\neq \Diamond$	-	$ubVarCard(r_2.ttav_1, TTAT_1, r_2) = 1$	1
r_2	$r_2.av_1$	$TTAA_1$	$\neq \Diamond$	-	$ubVarCard(r_2.ttav_1, TTAA_1, r_2) = \infty$	∞
r_3	$r_3.cv_1$	$NOAA_1$	$\neq \Diamond$	-	0	∞
r_3	$r_3.av_1$	$NOAA_1$	$\neq \Diamond$	-	0	1
r_3	$r_3.av_1$	TTA_1	$\neq \Diamond$	-	0	1
r_3	$r_3.av_1$	TAT_1	$\neq \Diamond$	-	$ubVarVard(r_3.tav_1, TAT_1, r_3) = 1$	∞
r_3	$r_3.tv_1$	TAA_1	$\Diamond$	1	-	1
r_3	$r_3.tv_1$	$VDAD_1$	$\Diamond$	1	-	1
r_3	$r_3.dv_1$	$VDAV_1$	$\Diamond$	1	-	∞
r_4	$r_4.c0v_1$	$PAAA_1$	$\neq \Diamond$	-	$ubVarCard(r_4.paav_1, PAAA_1, r_4) = \infty$	∞
r_4	$r_4.cae0v_1$	$PAAC_1$	$\Diamond$	1	-	1
r_4	$r_4.cae0v_1$	AAA_1	$\Diamond$	1	-	1
r_4	$r_4.cav_1$	$AAAE_1$	$\neq \Diamond$	-	$ubVarCard(r_4.caav_1, AAAE_1, r_4) = 2$	∞
r_5	$r_5.c0v_1$	$PSAG_1$	$\neq \Diamond$	-	$ubVarCard(r_5.psav_1, PSAG_1, r_5) = \infty$	∞
r_5	$r_5.gv_1$	$PSAC_1$	$\Diamond$	1	-	1
r_5	$r_5.gv_1$	$GCAC_1$	$\Diamond$	1	-	1
r_5	$r_5.c1v_1$	$GCAG_1$	$\neq \Diamond$	-	$ubVarCard(r_5.gcav_1, CGAG_1, r_5) = \infty$	∞

Table 5.9: The results of the application of Theorem 4.2.2.

As one can see from Table 5.9 all the calculated values for potentially created maximal cardinalities are less or equal to the upper bounds from the UML metamodel.

Since we have already shown that r is applicable we now can state that Theorem 4.2.6 does hold and our rule set is metamodel conform.

6 Conclusion and Future Work

In this work we presented a language with a graphical notation that allows one to specify very intuitive rules for the transformation of object oriented models. The strengths of the introduced BOTL language are its unambiguous formal definition and the possibility to prove the properties of applicability and metamodel conformance on the basis of a given rule set and its metamodels. This allows one to define rules and ensure that they are applicable for any given input model that is conform to the source metamodel. Further one can be sure that a rule set will always produce valid outputs, i.e. model that are conform to the given target meta model.

Since BOTL allows the transformation of arbitrary object oriented models there is a great potential application area for the language. Obviously the BOTL offers a lot of advantages over existing transformation languages when used for the integration of data from different applications, as indicated by our running example. Among our main interests is its application in the area of software engineering. So the project AUTOMOTIVE uses the BOTL for a special case of application integration: the integration of a CASE tool chain. Within the project KOGITO the language is used to formally prove the correctness of the mapping of various formally founded description techniques into a common conceptual development model. Further BOTL allows to specify the mapping of development models at different abstraction layers as proposed by the Model Driven Architecture (MDA) approach of the OMG. Thus the BOTL may also be used as a technique to verify development steps in a model-based software engineering approach.

A major ongoing work in this field is a tool support for BOTL. We are currently developing a prototype that allows a user to specify metamodels and rule sets with an extended CASE tool. This specified rule sets can be exported as XML documents that are read by a rule verifier called Neck. Neck analyzes a rule set with its metamodels according to our verification techniques for applicability and metamodel conformance. If the rules do not have the desired properties a XML document with a detailed error message is generated sent to the graphical editor and presented to the user. In the case that the given rule set did pass the tests it gets an approval annotation and is sent to a code generator. This code generator generates the code that performs the specified model transformations with internal representations of object models. We develop different plugins that the generated code may use. These allow the transformation code to consume and produce different object models like e.g. Java models, .NET models, or XMI documents.

Though BOTL is already a mature language that is ready to use we still plan to enhance the formalism to provide more powerful verification techniques for the proof of desired properties of rule sets and a more fine grained selection of source fragments that also considers desired attribute values. Further we plan to work out methodological guidelines that help a user to write

effective, extensible and readable rules for model transformations. All in all we are convinced that the presented work does form a solid base for many subsequent research activities in the field of object oriented model transformation.

Finally we would like thank the Bayrische Forschungsstiftung and the Bundesministerium für Bildung und Forschung for funding our work in FORSOFT II and KOGITO. Further we thank our colleagues Bernhard Schätz and Heiko Lötzbeyer for the helpful advices and fruitful discussions.

Bibliography

[Ake00] David H Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent at Canterbury, 2000.

[aut03] Automotive homepage. `http://www.forsoft.de/automotive`, 2003.

[BB95] Mohamed Bouneffa and Nacer Boudjlida. Managing schema changes in object-relationship databases. In M.P. Papazoglou, editor, *OOER 95 Object-Oriented and Entity-Relationship Modeling*, number 1021 in LNCS. Springer, 1995.

[BM02] Peter Braun and Frank Marschall. Transforming Object Oriented Models with BOTL. In Paolo Bottoni and Mark Minas, editors, *International Workshop on Graph Transformation and Visual Modeling Techniques*, number 72.3 in ENTCS. Elsevier Science B. V., 2002.

[BP98] T. Bray and J. Paoli. *Extensible Markup Language (XML) 1.0*. W3C, 1998. Available at http://www.w3.org/TR/1998/REC-xml-19980210.

[Cas95] E. Casais. *Managing Class Evolution in Object-Oriented Systems*, volume Object-Oriented Software Composition of *The Object-Oriented Series*, chapter 8. Prentice Hall, 1995.

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In Paolo Bottoni and Mark Minas, editors, *International Workshop on Graph Transformation and Visual Modeling Techniques*, number 72.3 in ENTCS. Elsevier Science B. V., 2002.

[kog03] Kogito homepage. `http://www.kogito.org`, 2003.

[LCC94] Chien-Tsai Liu, Panos K. Chrysanthis, and Shi-Kuo Chang. Database Schema Evolution through the Specification and Maintenance of Change on Entities and Relationships. In P. Lucopoulos, editor, *Entity-Relationship Approach - ER'94*, number 881 in LNCS. Springer, 1994.

[Mic03] Sun Microsystems. Enterprise JavaBeans Technology. `http://java.sun.com/products/ejb/`, May 2003.

[MM03] J. Miller and J. Mukerji. Mda guide version 1.0, May 2003.

[Nag96] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Springer, 1996.

[OMG00] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 2.4, formal/00-10-33, Object Management Group (OMG), www.omg.org, 2000.

[OMG02] OMG. OMG Unified Modeling Language Specification. Technical report, Object Management Group (OMG), www.omg.org, 2002.

[ORM01] OMG Architecture Board ORMSC. Model driven architecture (mda), July 2001. Document number ormsc/2001-07-01.

[PBG01] Mikaël Peltier, Jean Bézivin, and Gabriel Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *Workshop on Transformations in UML (WTUML), Genova, Italy*, April 2001.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, number 903 in LNCS, pages 151–163. Springer, 6 1994.

[Sol00] Richard Soley. Model Driven Architecture. OMG White Paper Draft 3.2, Object Management Group (OMG), November 2000.

[vBBRS02] Michael v.d. Beeck, Peter Braun, Martin Rappl, and Christian Schröder. Automotive Software Development: A Model Based Approach. Number 2002-01-0875 in SAE Technical Papers Series. Society of Automotive Engineers, SAE, 2002.

[VW95] Vania M.P. Vidal and Marianne Winslett. A rigorous approach to schema restructuring. In M.P. Papazoglou, editor, *OOER 95 Object-Oriented and Entity-Relationship Modeling*, number 1021 in LNCS. Springer, 1995.

[XMI99] XML Metadata Interchange (XMI). OMG Document ad/99-10-02, 1999.

[XML02] XML Schema Home Page, 2002.

[XSL99] XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 11 1999.